technische universität dortmund

fakultät für informatik

Diplomarbeit

ILP-basierte Registerallokation zur Worst-Case Execution Time Minimierung

Norman Schmitz

29. Juni 2010

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

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Informatik 12
Arbeitsgruppe Entwurfsautomatisierung für
Eingebettete Systeme
http://ls12-www.cs.tu-dortmund.de/daes/index.html

GERMANY · D-44221 DORTMUND

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel verwendet, sowie Zitate kenntlich gemacht habe.

Dortmund, den 29. Juni 2010

Norman Schmitz

Vorwort

Ich möchte mich an dieser Stelle bei allen Menschen bedanken, die mich im Laufe meines Studiums und insbesondere im Entstehungszeitraum dieser Diplomarbeit unterstützt haben. Danken möchte ich den Mitgliedern der Arbeitsgruppe Entwurfsautomatisierung für Eingebettete Systeme am Lehrstuhl 12, die immer ein offenes Ohr für meine Anfragen gehabt und mich unterstützt haben. Nennen möchte ich hier unter anderen Florian Schmoll, der einiges von seiner Zeit für meine Problemstellungen geopfert hat.

Ein ganz besonderer Dank geht an Dr. Heiko Falk für seine tadellose Betreuung und hilfreichen Anregungen innerhalb der letzten Monate. Seine großartige Unterstützung hat die Entstehung dieser Arbeit maßgeblich mitgetragen.

Auch meiner Freundin Jenny möchte ich danken, deren moralische Unterstützung mir in den letzten Monaten einen starken Rückhalt gegeben hat. Nicht zuletzt möchte ich meinen Eltern danken, die mir Zeit meines Studiums nach allen Kräften beigestanden und dieses erst ermöglicht haben.

Dortmund, den 29. Juni 2010

Norman Schmitz

Zusammenfassung

Im Bereich Eingebetteter Echtzeitsysteme spielt die Worst-Case Execution Time (WCET) eine herausragende Rolle innerhalb des Prozesses des Hardware-/Software-Codesigns. Gelingt es die WCET der verwendeten Software zu reduzieren, kann die benötigte Hardware entsprechend kleiner dimensioniert werden. Auf diese Weise können Herstellungs- und Betriebskosten in erheblichem Maße eingespart werden. Diese Diplomarbeit beschäftigt sich mit der WCET-Optimierung im Rahmen der Registerallokation und baut dabei auf dem Verfahren aus [Sch08] auf, welches den längsten Ausführungspfad (WCEP) innerhalb eines ganzzahligen Optimierungsproblems modelliert und dabei noch einige Schwächen offenbart. Ziel dieser Arbeit ist deshalb vor allem die Steigerung der Güte des bisherigen Verfahrens. Zu diesem Zweck wird das Pipelineverhalten der Zielarchitektur in die Modellierung integriert. Außerdem werden eine ganze Reihe weiterer Anpassungen vorgenommen.

Inhaltsverzeichnis

„Im Bereich der echtzeitfähigen Systeme ist eine Argumentation, die auf durchschnittlicher Leistungsfähigkeit aufbaut, nicht akzeptabel.“

Peter Marwedel
[Mar08, Seite 4]

1 Einleitung

Blicken wir genauer auf viele Dinge unseres täglichen Lebens, so stellen wir fest, dass uns eine große Anzahl von Computersystemen umgibt. Diese sind auf den ersten Blick oft nicht direkt erkennbar, und sie sind in allen möglichen Gerätschaften verbaut. Sie werden deshalb **Eingebettete Systeme** genannt und von Marwedel wie folgt definiert.

> *„Eingebettete Systeme sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind, und die normalerweise nicht direkt vom Benutzer wahrgenommen werden.“* [Mar08, Seite 1]

Sie unterstützen die Funktionalität des Produktes und sollen vom Kunden ansonsten nicht wahrgenommen werden. Je nach Einsatzgebiet steuern oder regeln sie die Funktionen des Produktes oder überwachen dessen Umwelt. Sie können aber auch bestimmten Aufgaben, wie der Daten- oder Signalverarbeitung, dienen. Durch den technischen Fortschritt, insbesondere einem Trend zur Miniaturisierung, können Eingebettete Systeme inzwischen vielfach eingesetzt werden. Sie finden sich an verschiedensten Stellen, beispielsweise:

- im Fahrzeugbau
- in Telekommunikationsgeräten
- in Haushaltsgeräten
- in der Unterhaltungselektronik
- in der Luft- und Raumfahrt
- in Produktionsmaschinen
- in der Robotik
- in der Medizintechnik
- in intelligenten Systemen (z. B. Gebäude)
- im militärischen Bereich

Die Bedeutung Eingebetteter Systeme steigt stetig, und längst haben sie herkömmliche Personal Computer zahlenmäßig hinter sich gelassen, weshalb eine Reihe

von Begriffen wie der der Post-PC Era entstanden sind. Ihre Besonderheiten gegenüber herkömmlichen Personal Computern drücken sich vor allem in den Anforderungen aus, die man an sie stellt. Sie sollen unter anderem verlässlich und effizient arbeiten. Zumeist wird an sie auch der Anspruch gestellt, dass sie **echtzeitfähig** sein sollen, was bedeutet, dass sie ihre Berechnungen innerhalb bestimmter Zeitschranken abschließen müsssen. Die maximale Ausführungszeit der verwendeten Programme muss deshalb bekannt sein. Effizienz kann sich im Bereich Eingebetteter Systeme an verschiedenen Kriterien orientieren. Folgende Eigenschaften können die Entwürfe der Hardware und zugehörigen Software beeinflussen, die im Regelfall parallel geschehen, und deshalb den Begriff des **Hardware-/Software-Codesigns** geprägt haben.

- Energieverbrauch
- Codegröße
- Laufzeit
- Gewicht
- Preis

Je nach verwendeten Programmen und gestellten Anforderungen kann die benötigte Hardware angepasst werden, um Kosten zu sparen. Die Software wird zu diesem Zweck häufig hinsichtlich eines Kriteriums oder mehrerer Kriterien optimiert.

1.1 Motivation

Die Software Eingebetteter Systeme wird vielfach durch Standardcompiler übersetzt, die oft billig und frei zugänglich sind. Dabei sollten besser speziell angepasste Compiler benutzt werden, da Eingebettete Systeme besondere Anforderungen stellen. So können etwa die speziellen Hardware-Eigenschaften ausgenutzt werden, um effizienten Programmcode zu erzeugen. Die benötigte Übersetzungszeit spielt im Vergleich zum möglichen Optimierungspotenzial eine untergeordnete Rolle. Verschiedene im vorherigen Abschnitt aufgezählte Eigenschaften können das Ziel von Optimierungen sein. Diese Diplomarbeit beschäftigt sich im Folgenden hauptsächlich mit der Optimierung der Laufzeit. Üblicherweise wird Laufzeitoptimierung mit der Reduzierung der durchschnittlichen Rechenzeit gleichgesetzt, was auch Hennessy und Patterson mit ihren Prinzipien geprägt haben.

> *„Make the Common Case Fast“* [HP03, Seite 39]
> *„Focus on the Common Case“* [HP07, Seite 39]

Viele Systeme im Bereich der Eingebetteten Systeme müssen **Echtzeitbedingungen** erfüllen, für sie wäre das Verharren auf der Reduzierung der durchschnittlichen Rechenzeit jedoch ein Denkfehler. Viel entscheidender ist hier eine andere Größe,

nämlich die der maximalen Ausführungszeit der eingesetzten Programme, die sogenannte **Worst-Case Execution Time (WCET)**. Die WCET beeinflusst die Dimensionierung der benötigten Hardware-Bestandteile und nimmt damit direkt Einfluss auf die Kosten des Eingebetteten Systems.

Die einzelnen Programme oder Programmbestandteile in einem **Eingebetteten Echtzeitsystem** müssen ihre Berechnungen innerhalb bestimmter Zeitspannen abschließen, da es ansonsten zu Qualitätseinbußen, Sachschäden oder gar Personenschäden kommen kann. Oftmals sind Echtzeitsysteme sicherheitskritisch, weil sie mit ihrer Umgebung verbunden sind und unmittelbaren Einfluss auf diese ausüben. Die Einhaltung von Zeitschranken und die korrekte Berechnung von Ergebnissen sind deshalb in solchen Systemen in ihrer Bedeutung gleichzusetzen. Betrachtet man beispielhaft die Airbagsteuerung eines Fahrzeuges, so kann die Auslösung zum optimalen Zeitpunkt entscheidend sein und eine präzise Einhaltung der Zeitschranken voraussetzen. Wird dieser Zeitpunkt verpasst, kann dies folgenschwer für die Insassen sein. Ein solches Echtzeitsystem wird daher auch als **hartes Echtzeitsystem** bezeichnet, für das Kopetz folgende Definition formuliert.

> *„Dies sind Systeme, in denen die Nichteinhaltung einer festgelegten Deadline zu katastrophalen Konsequenzen führen kann.“* (übersetzt aus dem Englischen) [Kop00, Seite 204]

Alle übrigen Echtzeitsysteme werden als weich bezeichnet. So kann beispielsweise die Nichteinhaltung von Zeitbeschränkungen bei einer Videoübertragung zu Aussetzern oder Sprüngen in der Wiedergabe führen, die in einigen Anwendungsgebieten durchaus akzeptabel sein können. Die Abgrenzung zwischen harten und weichen Echtzeitsystemen enthält eine gewisse Unschärfe, da eine Einstufung dem eigenen subjektiven Ermessen obliegt. Letztere Systeme umschreibt Kopetz wie folgt.

> *„Dies sind Systeme, in denen die Nichteinhaltung einer festgelegten Deadline die Brauchbarkeit des Ergebnisses reduziert, aber nicht zu erheblichen finanziellen Einbußen führt.“* (übersetzt aus dem Englischen) [Kop00, Seite 204]

Die Unterscheidung in weich oder hart hängt nie von der jeweiligen verwendeten Hardware, sondern immer von der Anwendung ab. Während des Entwurfs eines Echtzeitsystems müssen deshalb alle Effekte berücksichtigt werden, die Einfluss auf die maximale Ausführungszeit nehmen, damit die Einhaltung der Zeitschranken garantiert werden kann. Prinzipiell stehen zwei Lösungswege zur Verfügung, um die Einhaltung zu bewerkstelligen. So kann die Leistung der Hardware-Komponenten höher dimensioniert werden, was die Herstellungs- und Betriebskosten in die Höhe treiben kann. Der zweite und elegantere Weg ist die Optimierung der verwendeten Programme. Am Lehrstuhl 12 der Fakultät für Informatik der Technischen Universität Dortmund wird deshalb seit Jahren unter anderem Forschung im Bereich WCET-optimierender Verfahren betrieben, in deren Rahmen der WCET-optimierende Compiler **WCC** (Kapitel 3) entstanden ist.

> *„In harten Echtzeitsystemen ist eine niedrige WCET von größter Bedeutung, während die durchschnittliche Rechenzeit von geringem Interesse ist.“*
> (übersetzt aus dem Englischen nach Kopetz) [Kop00, Seite 210]

Eine möglichst geringe WCET und ein genaues Wissen über diese erlauben es, die Hardware derart an die Anwendung anzupassen, dass die Kosten niedrig gehalten werden können. Ein großes Potenzial zur Minimierung der WCET weist dabei die **Registerallokation** auf (Abschnitt 2.6), sie ist deshalb Thema dieser Diplomarbeit. In diesem Verfahren werden die Variablen eines zu übersetzenden Programmes den einzelnen Speicherstellen innerhalb des Prozessors zugeordnet, die **Register** heißen.

1.2 Zielstellung

Innerhalb des WCET-optimierenden Compilers WCC existieren verschiedene Registerallokatoren, unter anderem zwei, die eine WCET-Minimierung zum Ziel haben. Einer dieser Allokatoren wurde 2008 von Schmoll [Sch08] im Rahmen seiner Diplomarbeit entwickelt. Er wird deshalb in Kapitel 5 ausführlich erläutert. Dieses Verfahren ist **ILP-basiert** (Abschnitt 2.3) und benutzt geschätzte WCET-Daten, die aus einer vorhergehenden Analyse stammen. Als Zielarchitektur für die Übersetzung der Programme wird der **TC1796** (Kapitel 4) der Firma Infineon verwendet. Die gemessenen Ergebnisse dieses Allokators bzgl. der WCET sind jedoch wenig zusagend, vor allem auch im Vergleich zu anderen Allokatoren innerhalb des WCC.

Ziel dieser Diplomarbeit ist eine Steigerung der Güte der erzeugten Ergebnisse des genannten Registerallokators, da insbesondere im Kontext des WCC und der Eingebetteten Systeme die WCET eine herausragende Rolle spielt. Auf Basis der bestehenden Struktur des Registerallokators von Schmoll sollen deshalb folgende Erweiterungen respektive Veränderungen durchgeführt werden.

- Durch eine genauere Ermittlung der WCET_{est}-Werte soll die Präzision des ILP-Modells erhöht werden. Dazu sollen Eigenschaften der Prozessorpipeline der Zielarchitektur explizit berücksichtigt werden.
- Die Kosten von möglichen einzufügenden Spillinstruktionen innerhalb des ILPs sollen individuell modelliert werden, um die Eigenschaften der Prozessorpipeline der Zielarchitektur direkt mit ins Kalkül zu ziehen.
- Durch zusätzliche Erweiterungen wie der Umsetzung einer **Copy-Elimination** soll die Registerallokation optimiert werden.

1.3 Verwandte Arbeiten

Aufgrund des großen Optimierungspotenzials durch die Registerallokation, aber auch aufgrund der Komplexität des Registerallokationsproblems, existiert eine Viel-

zahl verschiedener Arbeiten zu diesem Thema. Die Modellierung ganzzahliger Optimierungsprobleme und deren Lösung durch Computerprogramme sind mittlerweile zu einem Standardverfahren für Zuordnungsprobleme geworden und werden in Abschnitt 2.3 näher erläutert. Die WCET wurde bis jetzt erst in wenigen Arbeiten gezielt optimiert. Zumeist beschäftigen sich die bzgl. der WCET publizierten Arbeiten explizit mit der Analyse und der Abschätzung ebendieser. Sie sollen hier nicht näher erwähnt werden, aber einen Überblick über diese Verfahren bieten [KP05, WEE+08, LGZ+09]. Im Folgenden werden einige Verfahren aus dem Bereich der Registerallokation und verschiedene WCET-optimierende Verfahren vorgestellt, um eine Einordnung dieser Diplomarbeit in den Bereich bereits erschienener Forschungsarbeiten aufzuzeigen.

1.3.1 Registerallokationsverfahren

Die Grundlage dieser Diplomarbeit stellt der ILP-basierte Registerallokator von Schmoll [Sch08] dar, der die WCET des zu übersetzenden Programmes minimieren soll. Das Registerallokationsproblem wird in Abschnitt 2.6 allgemein vorgestellt. Zudem werden hier auch der von Chaitin [Cha81] erstmals implementierte graphfärbungsbasierte Ansatz und dessen Weiterentwicklungen [CH90, BCT94, Fal09] erläutert. Letzterer wurde ebenfalls am Lehrstuhl 12 entwickelt und minimiert die WCET mit Hilfe von Heuristiken. Auch einige ILP-basierte Ansätze [GW96, KW98, AG01, FW02, Fu07] werden angeführt. Darunter befindet sich insbesondere der von Schmoll als Vorlage verwendete Ansatz von Goodwin und Wilken [GW96], der deshalb in Abschnitt 5.1 im Detail beschrieben wird.

Es gibt weitere Ansätze, das Registerallokationsproblem abzubilden. Zu nennen ist dabei das Linear-Scan-Verfahren, welches zuerst in [THS98, PS99] vorgestellt wurde und seitdem auch Inhalt zahlreicher anderer Studien war [WM05, SB07, QPP08, WF10]. Es operiert auf einer linearen Ordnung der Lebenszeiten der verwendeten Variablen. Dieses greedyartige Verfahren zeichnet sich durch seine geringe Laufzeit aus, die sich in der Beobachtung als nahezu linear darstellt, aber es erzeugt oftmals schlechte Ergebnisse. In [SE02] und [HS06] wird die Möglichkeit der Abbildung auf ein PBQP (Partitioned Boolean Quadratic Problem) beschrieben, das wiederum eine besondere Form des Quadratischen Zuordnungsproblems ist. Die Komplexität dieser Variante wird mit $\mathcal{O}(|V|K^3)$ angegeben, wobei V die Variablen sind und K die Anzahl der Register ist. Das Finden einer optimalen Lösung kann lange dauern, weswegen auch hier häufig Heuristiken benutzt werden. Viele Verfahren wie z. B. in [Nin93] teilen das Registerallokationsproblem in Teilprobleme auf, die leicht zu lösen sind. Durch die Verwendung von Heuristiken stellt das Gesamtergebnis i. d. R. jedoch keine optimale Lösung mehr dar.

Auf Grundlage der vorgestellten Verfahren können beliebige Optimierungsziele verfolgt werden. Zhang et al. [ZHC02] versuchen gezielt die Energieaufnahme wäh-

rend der Programmausführung zu reduzieren. In [LG97] berücksichtigen Lueh und Gross die Kosten von Funktionsaufrufen.

1.3.2 WCET-Optimierungsverfahren

Im Bereich der WCET-optimierenden Verfahren sollen zunächst einmal jene erwähnt werden, die im Rahmen der Entwicklung des Compiler-Frameworks WCC [FLT06] in den letzten Jahren am Lehrstuhl 12 entstanden sind. Der WCC bezieht als erster Compiler überhaupt $WCET_{est}$-Werte in den Übersetzungsprozess mit ein und wird in Kapitel 3 ausführlich dargestellt. Um WCET-basierte Optimierungen durchführen zu können, wurde der WCC um die Möglichkeiten der Annotation des Programmcodes mit **Flow-Facts** [Sch07] (Abschnitt 2.2), der automatischen Schleifenanalyse [Cor08] und der Transformation von WCET-Daten in die High-Level-Zwischendarstellung [Ged08] erweitert. Die WCET-optimierenden Verfahren innerhalb des WCC werden in zwei Gruppen unterteilt hier aufgeführt. Als erstes sind nachfolgend die hardwarenahen Low-Level-Optimierungen aufgelistet:

- Die zuvor schon erwähnten WCET-basierten Registerallokatoren [Sch08, Fal09]
- Instruction Cache Locking [FPT07]
- Cache-basiertes Procedure Positioning [LFM08]
- Statische Scratchpad-Allokation von Daten [Rot08]
- Cache-Partitionierung für Multi-Task-Systeme [PLM09]
- Statische Scratchpad-Allokation von Programmcode [FK09]
- Loop Invariant Code Motion mit Hilfe von maschinellem Lernen [LSMM10]

Nun folgen die hochsprachenorientierten High-Level-Optimierungen, die der WCC zur Verfügung stellt:

- Function Specialization durch Procedure Cloning [LFMT08]
- Function Inlining [LGMM09]
- Loop Unswitching [LGM09]
- Loop Unrolling [LM09]
- Superblock-basierte Optimierungen [Kel09]

Da die übliche Einteilung in Optimierungsstufen bei Eingebetteten Systemen oft unzureichend ist, wurde im WCC die Möglichkeit für multikriterielle Optimierungen geschaffen [LPF+10]. Auf diese Weise kann die WCET in Kombination mit anderen Zielen optimiert werden.

In [SMRC05] wird von Suhendra et al. ebenfalls ein Verfahren zur Scratchpad-Allokation der Daten vorgestellt, das die WCET minimieren soll. Dieses Verfahren dient als Grundlage der in dieser Diplomarbeit behandelten Registerallokation und wird deshalb in Abschnitt 5.2 näher erläutert. Zhao et al. [ZKW+06] stellten

eine Reihe von Verfahren zur WCET-Minimierung vor, wie z. B. Loop Unrolling und Superblock-basierte Optimierungen. Puaut und Pais [PP07] haben ebenfalls die WCET über Instruction Cache Locking minimieren können.

1.4 Aufbau der Arbeit

Zunächst werden in **Kapitel 2** einige Begriffe und Verfahrensweisen beschrieben, die von grundlegender Bedeutung für diese Diplomarbeit sind. Darunter befinden sich die Begriffe des Kontrollflussgraphen (Abschnitt 2.1), der Flow-Facts (Abschnitt 2.2), der ganzzahligen Optimierung (Abschnitt 2.3). Im Abschnitt 2.4 wird die Worst-Case Execution Time vorgestellt. Wie diese durch Analysen bestimmt werden kann, zeigt Abschnitt 2.5. Zudem wird das Verfahren der Registerallokation allgemein im Abschnitt 2.6 vorgestellt.

In **Kapitel 3** wird der zuvor schon erwähnte WCET-optimierende Compiler WCC vorgestellt, der am Lehrstuhl 12 entwickelt wird. Nachdem auf die Struktur eingegangen wurde, werden unter anderem die Registerallokationsverfahren des WCC dargestellt. Zum Schluss dieses Kapitels wird in Abschnitt 3.3 das Analyseprogramm a^3 beschrieben.

Der für diese Diplomarbeit als Zielarchitektur verwendete TC1796 der Firma Infineon wird in **Kapitel 4**, vor allem auch in Bezug auf sein Pipelineverhalten, erläutert.

Bevor die eigene Arbeit vorgestellt werden soll, wird noch einmal die Verfahrensweise des Registerallokators von Schmoll in **Kapitel 5** umschrieben. An dieser Stelle sei besonders auf die **Abschnitte 5.4** und **5.5** hingewiesen, die die Schwachstellen der bisherigen Implementierung und die ihnen zugedachten Lösungsansätze auflisten.

Den Hauptteil dieser Diplomarbeit stellen die beiden in **Kapitel 6** präsentierten Verfahren dar. Zum einen wird hier die statische WCET-Analyse durch eine architekturbezogene Pipelineanalyse unterstützt (**Abschnitt 6.4**), und zum anderen wird das ILP um individuelle Kostenmodellierungen für jede Spillinstruktion erweitert (**Abschnitt 6.5**).

Zusätzlich integrierte Erweiterungen werden in **Kapitel 7** einzeln dargestellt. Diese umfassen die Verwendung des Scratchpad-Speichers während der Ermittlung der WCET_{est}-Daten (Abschnitt 7.1), eine Variation der Vorallokation (Abschnitt 7.2), die Integration der Copy-Elimination in das ILP (Abschnitt 7.3) sowie die Kombination der Zielfunktion mit codegrößenoptimierenden Elementen (Abschnitt 7.4).

Die Auswirkungen der vorgenommenen Änderungen und Erweiterungen werden durch die Ergebnisse des Benchmarkings in **Kapitel 8** aufgezeigt. Insbesondere wird das

Verhalten des Registerallokators in Bezug zu dem anderer im WCC eingesetzter Verfahren untersucht.

Eine Bewertung der Ergebnisse gibt das **Kapitel 9**. Abschließend wird ein Ausblick auf mögliche zusätzliche Erweiterungen in **Kapitel 10** gegeben.

„Aufgrund der zentralen Rolle, die die Registerallokation bei der Beschleunigung des Codes und der Nutzbarmachung anderer Optimierungen spielt, ist sie eine der wichtigsten – wenn nicht die wichtigste – der Optimierungen.“

John L. Hennessy und David A. Patterson
übersetzt aus dem Englischen [HP07, Seite B-26]

2 Grundlagen

Für das Verständnis dieser Diplomarbeit wird das Wissen über einige grundlegende Begriffe und Verfahrensweisen benötigt, die dieses Kapitel näher erläutert und definiert. Alle nachfolgenden Kapitel setzen das hier dargestellte Wissen als bekannt voraus. Nacheinander werden die Begriffe des Kontrollflussgraphen (Abschnitt 2.1), der Flow-Facts (Abschnitt 2.2) und der Ganzzahligen Optimierung (Abschnitt 2.3) eingeführt. Im Anschluss wird der für diese Diplomarbeit bedeutende Begriff der maximalen Ausführungszeit WCET in Abschnitt 2.4 definiert. Nachfolgend wird im Abschnitt 2.5 gezeigt, wie die WCET durch Analysen approximiert werden kann. Eine Zusammenfassung zum Thema des Registerallokationsproblemes wird abschließend im Abschnitt 2.6 dieses Kapitels gegeben.

2.1 Kontrollflussgraph

Ein **Kontrollflussgraph** (auch CFG, von Control Flow Graph) ist ein spezieller, gerichteter Graph in der Informatik. Er wird aus einem Programm abgeleitet und stellt den Kontrollfluss dieses Programmes dar. Mit Hilfe eines CFG können **Kontrollflussanalysen** (CFA für Control Flow Analysis) und Optimierungen auf einem Programm angewendet werden. Da bei der Erstellung eines Kontrollflussgraphen keine Auswertung von bedingten Sprüngen vorgenommen werden kann, werden alle nachfolgend möglichen Kontrollflüsse durch Verzweigungen an den bedingten Sprüngen ausgedrückt. Schleifen sind als Zyklen im CFG erkennbar. Die einzelnen Knoten und Kanten eines CFG können mit beliebigen Informationen annotiert werden.

Definition 2.1 (CFG - Kontrollflussgraph) *Ein CFG besteht aus einem Tupel $G\langle V, E, r\rangle$[Muc97]. Die Menge V besteht aus Knoten, die z. B. entweder Instruktionen, Basisblöcke oder Funktionen repräsentieren. Diese Knoten sind durch die gerichteten Kanten $u \rightarrow v$ aus der Menge $E\colon V \rightarrow V$ miteinander verbunden, wenn v unmittelbar nach der Ausführung von u ausgeführt werden kann. Der Knoten r wird Wurzelknoten oder Quelle genannt und ist der Startpunkt des dargestellten Programmes. Er ist der einzige Knoten aus V ohne Vorgänger. Für die Nachfolgeknoten Succ und die Vorgängerknoten Pred eines Knoten $u \in V$ gilt:*

$$Succ(u) = \{v \in V \mid \exists e \in E,\, e = u \rightarrow v\} \tag{2.1}$$

$$Pred(u) = \{v \in V \mid \exists e \in E,\, e = v \rightarrow u\} \tag{2.2}$$

Zusätzlich existiert auch eine ausgezeichnete Menge von Endknoten, in denen das Programm terminiert, diese werden auch Blätter oder Senken genannt. Ein CFG kann verschiedene Ausprägungen haben. Enthält er Funktionen und deren Aufrufrelationen, so spricht man von einem Call-Graph. Er kann ebenso **Basisblöcke** oder einzelne Instruktionen enthalten und z. B. zur WCET-Analyse eingesetzt werden.

Definition 2.2 (Basisblock) *Einen Basisblock (kurz „BB“) definiert [Muc97] als eine maximale Sequenz von Instruktionen ($I_1, ..., I_n$), so dass gilt:*

- *Der Basisblock wird nur über die erste Instruktion I_1 betreten.*
- *Der Basisblock wird nur über die letzte Instruktion I_n verlassen.*

Dementsprechend sind die CFGs von Basisblöcken per Definition zyklenfrei. Pfade, die bei der Ausführung des Programmes niemals durchlaufen werden können, weil z. B. eine bedingte Anweisung niemals erfüllt wird, heißen **unausführbare Pfade (Infeasible Paths)**. Im Zusammenhang mit CFGs treten die Begriffe der **Lebendigkeitsanalyse**, **Dominanzrelation** und **Rückwärtskante** auf.

Definition 2.3 (Variable benutzen) *Eine Instruktion benutzt (kurz „use“) eine Variable genau dann, wenn der Variableninhalt bei der Abarbeitung ausgelesen wird.*

Definition 2.4 (Variable definieren) *Eine Instruktion definiert (kurz „def“) eine Variable genau dann, wenn ihr Inhalt bei der Abarbeitung verändert wird.*

Mit Hilfe der Definitionen 2.3 und 2.4 kann jetzt der Begriff der Lebendigkeit definiert werden. Eine Lebendigkeitsanalyse (LTA für Live Time Analysis) berücksichtigt die Lebendigkeit aller Variablen und liefert für jeden Knoten des CFG die Mengen LiveIn und LiveOut.

Definition 2.5 (Lebendigkeit) *Nach [App04] ist eine Variable an einer Kante lebendig (live), wenn von dort aus ein Pfad zu einer Benutzung existiert, auf dem diese Variable nicht neu definiert wird. Des Weiteren ist eine Variable in einem Knoten live-in, wenn mindestens eine der eingehenden Kanten lebendig ist. Analog dazu ist eine Variable in einem Knoten live-out, wenn mindestens eine der ausgehenden Kanten lebendig ist.*

Für einen Knoten n mit den Mengen der benutzen und definierten Variablen lassen sich die Mengen der Variablen bestimmen, die live-in und live-out sind.

$$LiveIn(n) = Use(n) \cup (Out(n) - Def(n)) \tag{2.3}$$

$$LiveOut(n) = \bigcup_{s \in Succ(n)} liveIn(s) \tag{2.4}$$

$$LiveOut(n)\, und\, LiveOut(n) : Knoten \rightarrow \mathcal{P}(Variablen)$$

Definition 2.6 (Dominanzrelation) *Führt jeder Pfad vom Startknoten r, der im Knoten v endet, über den Knoten u, so sagt man, dass der Knoten u den Knoten v dominiert (kurz „u dom v“).*

Definition 2.7 (Rückwärtskante) *Gelangt man im Laufe einer Tiefensuche innerhalb eines Graphen über einen Knoten v zu einem Konten u, der zuvor schon entdeckt wurde, so heißt die Kante von v nach u Rückwärtskante.*

Mit Hilfe dieser beiden Defintionen lassen sich auch die Begriffe des reduzierbaren und irreduzierbaren Graphen formulieren.

Definition 2.8 (Reduzierbarer Graph) *Ein Graph ist reduzierbar, wenn für jede Rückwärtskante $v \rightarrow u$ gilt, dass der Knoten u den Knoten v dominiert*

Definition 2.9 (Irreduzierbarer Graph) *Jeder nicht reduzierbare Graph ist irreduzierbar.*

2.2 Flow-Facts

In [KP05] werden Flow-Facts als Metadaten definiert, die Hinweise zum Kontrollflussverhalten des zugehörigen Programmes darstellen. Mit ihrer Hilfe kann die maximale Ausführungszeit eines Programmes abgeschätzt werden. Für eine möglichst genaue Abschätzung sind präzise Flow-Facts unabdingbar. Nachfolgend sind die möglichen Arten von Flow-Facts aufgelistet:

- obere Schranken für die Iterationszahlen von Schleifen
- Rekursionstiefen von Funktionen
- unausführbare Pfade (Infeasible Paths)
- Eigenschaften von Basisblöcken (z. B. Ausführungshäufigkeit)
- Abhängigkeiten von Programmteilen (z. B. Verhältnis der Ausführungshäufigkeit zweier Basisblöcke)
- Ausführungsreihenfolge
- möglicher Wertebereich von Eingabevariablen

Es gibt implizite und annotierte Flow-Facts. Die impliziten werden automatisch durch Analysen der Syntax und Semantik des Programmes gewonnen. Oft verwendete Methoden sind hierbei die Erkennung von Daten- und Kontrollflussabhängigkeiten, sowie die Abstrakte Interpretation und die Symbolische Ausführung. Diese können teilweise sehr komplex sein, wenn ein möglichst präzises Ergebnis für die Flow-Facts erreicht werden soll. Im optimalen Fall könnten alle benötigten Flow-Facts durch die impliziten Flow-Facts abgedeckt werden. Leider ist dies i. d. R. nicht für alle wünschenswerten Informationen möglich, da nur einfache Fälle von Analyseprogrammen erfasst und bewertet werden können. Dies folgt aus der Unentscheidbarkeit des Halte-Problems [Weg96], da die Dauer, die ein Programm in einer Schleife verweilt, auf das Halte-Problem reduziert werden kann und damit für beliebige Fälle unentscheidbar ist.

Um jedoch Flow-Facts im ausreichenden Maße zur Verfügung zu haben, müssen zusätzliche Informationen manuell ausgewertet und annotiert werden. Die Menge der impliziten und annotierten Flow-Facts sind dabei keineswegs disjunkt. Annotierte Flow-Facts können auf verschiedene Weisen zugänglich gemacht werden, etwa durch Erweiterung des Quellcodes, das Beifügen einer Datei oder eine direkte Eingabe. Zu beachten ist, dass die Flow-Facts während der Arbeitsschritte des Compilers konsistent gehalten werden. Vor allem bei Optimierungen oder den Übergängen zu anderen Zwischendarstellungen ändert sich die Struktur des Programmcodes, und die Flow-Facts bedürfen einer Anpassung.

Flow-Facts werden auch im WCC benutzt, um die WCET-Analyse mittels a^3 (Abschnitt 3.3) zu ermöglichen [Sch07]. Während in [KP05] so verfahren wird, dass sie an die Kanten des Kontrollflusses angehängt werden, knüpft der WCC sie an die einzelnen Programmbestandteile, wie z. B. Basisblöcke.

2.3 Ganzzahlige Optimierung

In der Ganzzahligen Optimierung [Kal02] werden reale Probleme auf mathematische Modelle mit einem beschränkten Zustandsraum abgebildet. Diese Modelle werden auch **ILP** (integer linear programming, manchmal auch nur LP oder IP) oder **Optimierungsproblem** genannt. Synonym werden für die Ganzzahlige Optimierung auch die Begriffe kombinatorische oder diskrete Optimierung verwendet.

Bis 1980 nur ein Randgebiet der Wissenschaft, gewann die Ganzzahlige Optimierung vor allem seit Anfang der 1990er Jahre immer mehr an Bedeutung und wird heute in zahlreichen Bereichen eingesetzt. Als Beispiele seien hier die Logistik, Produktionsplanung, Finanzwirtschaft, Kommunikationsbranche und Designabteilungen genannt. In der Informatik wird sie besonders für Zuordnungsprobleme und Reihenfolgeplanungen (Scheduling) verwendet. Die starke Verbreitung resultiert vor allem aus der Weiter- und Neuentwicklung der Lösungsalgorithmen und der Stei-

gerung der Rechenkapazitäten. In dieser Diplomarbeit wird der leistungsstarke und häufig eingesetzte ILP-Solver **CPLEX**[1] der Firma IBM ILOG verwendet.

Definition 2.10 (ILP) *ILP-Modelle bestehen aus einer Kostenfunktion und einer Menge J von Nebenbedingungen [Mar08]. Beide enthalten Verweise auf eine Menge $X = \{x_i\}$ von ganzzahligen Variablen und müssen lineare Funktionen ebendieser Variablen sein.*

$$C = \underbrace{\sum_{x_i \in X} a_i x_i}_{Kostenfunktion}, \ mit\ a_i \in \mathbb{R},\ x_i \in \mathbb{Z} \tag{2.5}$$

$$\forall j \in J\colon \underbrace{\sum_{x_i \in X} b_{i,j} x_i}_{Nebenbedingungen} \geq c_j, \ mit\ b_{i,j}, c_j \in \mathbb{R} \tag{2.6}$$

Bei geeigneter Wahl der Konstanten $b_{i,j}$ kann jedes $\leq$, $<$ oder $>$ in einer Nebenbedingung durch ein $\geq$ ersetzt werden.

Variablen, die nur die Werte Null und Eins annehmen können, heißen binäre Variablen oder Entscheidungsvariablen. Die Nebenbedingungen eines ILPs schränken den Lösungsraum ein. Durch das Finden einer optimalen Variablenbelegung wird die Zielfunktion des ILPs maximiert oder minimiert. In der Realität entspricht dies zumeist einer Gewinnmaximierung oder Kostenminimierung.

Zur Lösung von Optimierungsproblemen gibt es eine ganze Reihe von algorithmischen Ansätzen. Die am meisten verwendeten sind das Simplexverfahren und die Innere-Punkte-Methode. Beim Simplexverfahren wird ausgenutzt, dass der Lösungsraum einen konvexen Polyeder darstellt. Da die optimale Lösung in einer der Ecken liegen muss, werden diese iterativ abgelaufen. Welche Methode sich besser eignet, hängt von der Modellierung ab. Oft werden beide Verfahren kombiniert eingesetzt, zusammen mit einer Vorverarbeitung.

Ganzzahlige Optimierungsprobleme sind NP-vollständig. Entscheidend für die Komplexität eines einzelnen Optimierungsproblemes ist nicht die Anzahl der Variablen oder Nebenbedingungen, sondern die richtige Verwendung der Variablen und eine geeignete Formulierung. Eine geschickte Wahl der Zielfunktion kann das Ergebnis merklich beeinflussen. Bei einer schlechten Wahl kann jeder Punkt auf dem Rand des Polyeders eine optimale Lösung des ILP sein.

2.4 WCET - Worst-Case Execution Time

Die WCET (Worst-Case Execution Time) eines Programmes ist die obere Schranke für die Ausführungszeit über alle möglichen Eingaben und Ausgangszustände.

[1] http://www-01.ibm.com/software/integration/optimization/cplex/

Sie kann im Allgemeinen nicht automatisch bestimmt werden, weil man ansonsten in $\mathcal{O}(1)$ für beliebige Programme entscheiden könnte, ob sie terminieren und so das Halte-Problem lösen könnte, das nach [Weg96] unentscheidbar ist. Analog dazu existiert mit der **BCET** (Best-Case Execution Time) eine untere Schranke. Beide Schranken zusammen grenzen den Bereich aller möglichen Ausführungszeiten ein.

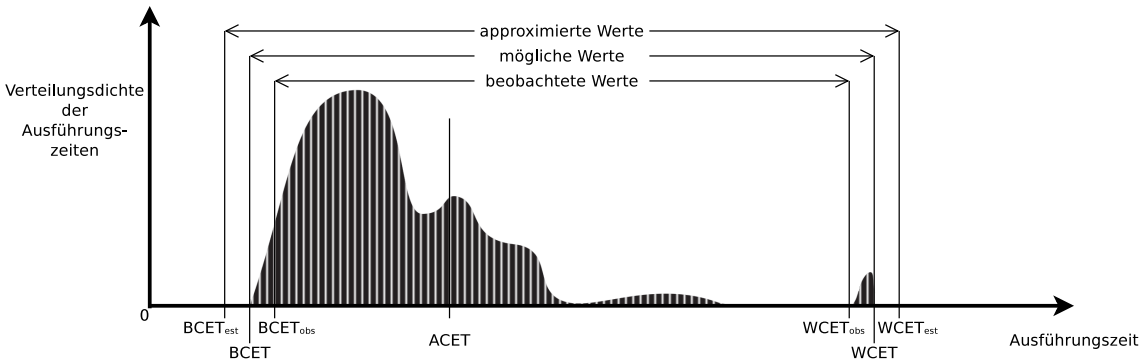


Abbildung 2.1: WCET nach [WEE+08]

Will man beide Werte approximieren, so unterschätzt man die BCET bzw. überschätzt die WCET, um sicher zu gehen. Auf diese Weise erhält man die Schätzwerte $\mathbf{BCET}_{est}$ (Estimated BCET) und $\mathbf{WCET}_{est}$ (Estimated WCET). Führt man das zu analysierende Programm unter verschiedenen Eingaben aus, so beobachtet man i. d. R. unterschiedliche Ausführungszeiten, die größer als die BCET und kleiner als die WCET sind. Diese Messwerte werden $\mathbf{BCET}_{obs}$ (Observed BCET) und $\mathbf{WCET}_{obs}$ (Observed WCET) genannt. Da sich verschiedene widersprüchliche Definitionen bzgl. der WCET verbreitet haben [Mar08], soll hier noch einmal erwähnt werden, dass für diese Diplomarbeit durchgängig die Definition aus [WEE+08] verwendet wird. Insbesondere sollen zwischen den vorherstehenden Begriffen folgende Bedingungen möglichst erfüllt sein:

$$\underset{\text{Sicherheit (safeness)}}{\mathrm{WCET} \leq \mathrm{WCET}_{\mathrm{est}}} \tag{2.7}$$

$$\underset{\text{Präzision (tightness)}}{\mathrm{WCET}_{\mathrm{est}} - \mathrm{WCET} \rightarrow \text{minimal}} \tag{2.8}$$

Noch zu erwähnen ist die durchschnittliche Ausführungszeit eines Programmes, die **ACET** (Average-Case Execution Time). Diese kann, wie die WCET und BCET, ebenfalls nicht exakt bestimmt, sondern nur angenähert werden. Das hauptsächliche Problem mit der Bestimmung der vorhergehenden Werte liegt darin, dass die Worst-Case- bzw. Best-Case-Eingabe und der Worst-Case- bzw. Best-Case-Startzustand nicht automatisch bestimmt werden können. Die Prüfung aller möglichen Eingaben in Kombination mit allen möglichen Startzuständen ist zudem i. d. R. nicht durchführbar.

2.4.1 WCEP - Worst-Case Execution Path

Liegen die $WCET_{est}$-Werte der einzelnen Instruktionen und Basisblöcke vor, kann der **WCEP** (Worst-Case Execution Path) bestimmt werden. Dieser ist der Pfad im CFG, der bei seiner Abarbeitung zur WCET führt. Er ist nicht fest und kann sich z. B. durch Optimierungen verschieben. Diese Instabilität des WCEP stellt die Abbildung 2.2 dar. Auf der linken Seite repräsentiert der linke Zweig den WCEP innerhalb des CFG. Durch eine Reduzierung der WCET des Basisblocks L3, auf der rechten Seite der Abbildung, ändert sich dieser und verschiebt sich auf den rechten Zweig.

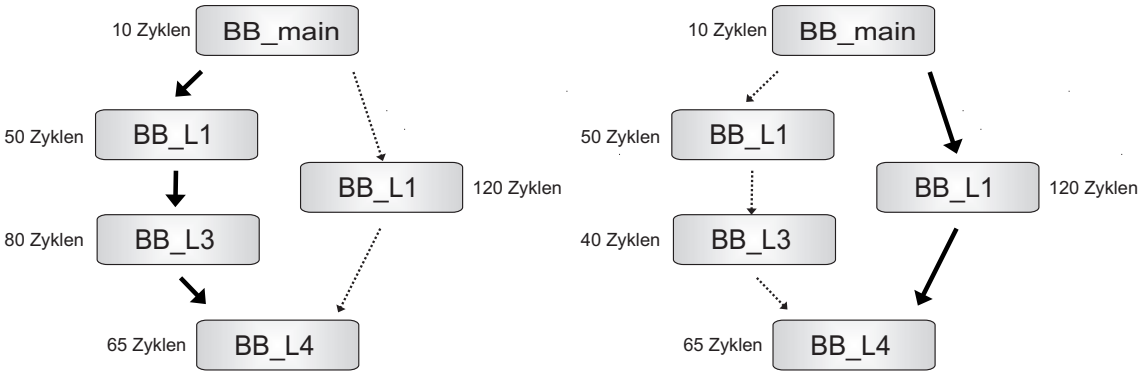


Abbildung 2.2: Instabilität des WCEP

In [WEE+08] werden drei Varianten genannt, mit deren Hilfe der WCEP analytisch bestimmt werden kann. Dazu müssen die $WCET_{est}$-Werte aller Basisblöcke bekannt sein.

IPET-Methode: Die IPET-Methode (Implicit Path Enumeration Technique) wurde von Li und Malik [LM95] entworfen und seitdem in vielen Arbeiten in erweiterter Form verwendet. Grundlage ist ein ILP, das alle möglichen Ausführungspfade modelliert und schlussendlich den WCEP und die WCET zugleich liefert. Es werden Variablen für jeden Basisblock und jede Kante des CFG erstellt, die die Ausführungshäufigkeit darstellen.

Die Nebenbedingungen orientieren sich an den Kirchhoffschen Gesetzen und stellen sicher, dass ein Basisblock nur so oft durchlaufen werden kann, wie er durch die eingehenden Kanten betreten und die ausgehenden Kanten verlassen wird. Diese Nebenbedingungen heißen strukturelle Nebenbedingungen. Die Quelle und die Senken des CFG erhalten jeweils eine Nebenbedingung, die sicherstellt, dass sie nur einmal durchlaufen werden können. Für Schleifen werden Nebenbedingungen eingefügt, die die obere Schranke für die Anzahl der Iterationen modellieren. Die Kostenfunktion setzt sich zusammen aus den Produkten der $WCET_{est}$ und der Ausführungshäufigkeit der einzelnen Basisblöcke, die aufsummiert werden und anschließend maximiert werden sollen. Anhand der Abbildung 2.3 wird die Modellierung nochmals beispielhaft verdeutlicht.

Start- und Endnebenbedingungen:

$X_{Start} = 1,\ X_{Ende} = 1$

Schleifennebenbedingungen:

$X_A \leq 100$

Strukturelle Nebenbedingungen:

$X_{Start} = X_{StartA}$
$X_A = X_{StartA} + X_{EA} = X_{AEnde} + X_{AB}$
$X_B = X_{AB} = X_{BC} + X_{BD}$
$X_C = X_{BC} = X_{CE}$
$X_D = X_{BD} = X_{DE}$
$X_E = X_{CE} + X_{DE} = X_{EA}$
$X_{Ende} = X_{AEnde}$

Zielfunktion:

$$WCET_{est} = max(X_A * WCET_{est}(A) + \\ ... + X_E * WCET_{est}(E))$$

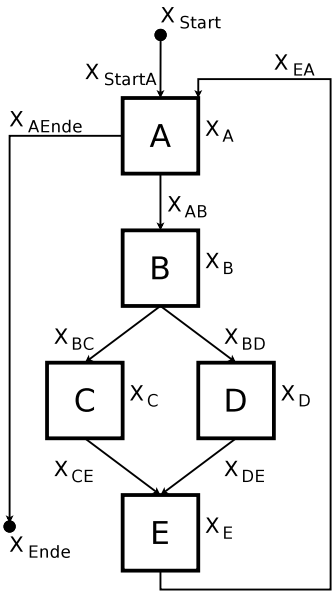


Abbildung 2.3: Beispiel eines ILPs der IPET-Methode nach [WEE+08]

Obwohl diese Modellierung NP-vollständig ist, liefert sie in der Praxis Ergebnisse in akzeptabler Zeit. Der große Vorteil dieser Modellierung besteht in der Flexibilität. So können nachträglich weitere Nebenbedingungen einfach hinzugefügt werden. Auf diese Weise können z. B. neue Flow-Facts integriert werden. Zudem lassen sich mit dieser Modellierung alle möglichen Kontrollflüsse abbilden, auch jene mit mehreren Schleifenausgängen.

Strukturbasierte Methode: Bei dieser Bottom-Up-Methode wird der Syntaxbaum des Programmes durch Traversierung anhand einer Grammatik von den Blättern her auf den Wurzelknoten zurückgeführt. Dabei werden sequentielle Programmstücke, bedingte Anweisungen und Schleifen nach und nach durch Ersatzknoten ausgetauscht. Die $WCET_{est}$ wird parallel nach den folgenden Regeln berechnet (T steht hier verkürzend für die $WCET_{est}$):

$$T(A) = T(S1) + T(S2) \quad \text{für A := seq(S1,S2)} \tag{2.9}$$

$$T(B) = T(Ausdruck) + max(T(S1) + T(S2)) \quad \text{für B := if(Ausdruck) S1 else S2} \tag{2.10}$$

$$T(C) = T(Ausdruck) + (T(Ausdruck) + T(Rumpf)) \cdot Iterationen \qquad (2.11)$$
$$\text{für C := loop(Ausdruck, Rumpf)}$$

Ein Nachteil dieser Methode ist, dass sie nachträglich nicht erweiterbar ist, z. B. um Flow-Facts. Einige Programmabläufe sind sogar nicht darstellbar, wie z. B. Schleifen mit mehreren Ausgängen. Dies ist allerdings nicht auf das Programm, sondern auf die Methode zurückzuführen, da andere Methoden diese Schwäche nicht aufweisen.

Pfadbasierte Methode: Hier werden alle möglichen Ausführungspfade einzeln untersucht. Dabei werden die Kosten der einzelnen Bestandteile berücksichtigt. Diese werden zusätzlich mit den Flow-Facts, wie z. B. Schranken für Schleifeniterationen, verrechnet. Als WCEP ergibt sich schlussendlich der Pfad, für den die Analyse die größte Ausführungsdauer geliefert hat. Der Vorteil dieser Methode liegt darin, dass Informationen alle Pfade betreffend anschließend zur Verfügung stehen und das Ergebnis die WCET sehr gut approximiert. Als Nachteil ist die mögliche große Rechenzeit zu nennen. Die Anzahl der möglichen Pfade wächst exponentiell zur Anzahl der Verzweigungen. Für Programme mit vielen Schleifen oder bedingten Anweisungen kann so der Suchraum schnell zu groß werden. Deshalb werden bei dieser Methode oftmals Heuristiken eingesetzt.

2.5 WCET-Analyse

Dieser Abschnitt beschäftigt sich mit der Analyse der WCET, da diese die maßgebliche Größe in dieser Diplomarbeit ist. Ein präzises Wissen über diese obere Schranke für die maximale Programmlaufzeit ist in Eingebetteten Systemen von großer Bedeutung. Die hohe Komplexität der heutigen Architekturen macht die Bestimmung dieses Wertes zu einem schwierigen Unterfangen. Die einzelnen Bestandteile und Funktionen, wie die Pipeline, Caches, Speicher oder Sprungvorhersage, verkomplizieren eine genaue Abschätzung des Verhaltens.

Im folgenden werden die zwei gängigen Prinzipien im Bereich der WCET-Analysen erläuert. Zunächst wird die statische WCET-Analyse und im Anschluss die messungsbasierte WCET-Analyse vorgestellt.

2.5.1 Statische WCET-Analyse

Bei statischen WCET-Analysen werden die $WCET_{est}$-Werte eines Programmes analytisch bestimmt, ohne dieses auszuführen [WEE+08]. Die Ergebnisse stellen i. d. R. sichere obere Schranken für die WCET dar, weil die Kosten der einzelnen Programmbestandteile überschätzt werden. Für diese Analysen werden abstrakte Modelle der

verwendeten Hardware benötigt, die in Abhängigkeit von der Architektur sehr komplex ausfallen können.

Zusätzlich ist zu beachten, dass im Bereich von Eingebetteten Systemen anstatt von Assemblerprogrammierung mittlerweile Hochsprachen wie C genutzt werden. Durch den Verzicht auf bestimmte Hochsprachenelemente könnte man die statische Analyse unterstützen, würde aber auf der anderen Seite Funktionalität einbüßen. Man behilft sich auf anderem Wege. Es ist eine gängige Praxis geworden, den Programmcode mit Kontrollflussinformationen, sogenannten Flow-Facts (Abschnitt 2.2), zu versehen. Die WCET-Analyse zieht diese heran, um die Vorhersage präzisieren zu können. Während die Programmierung immer mehr auf Hochsprachenebene geschieht, werden die WCET-Analysen ausschließlich auf Assemblerebene durchgeführt. Durch Optimierungsschritte und Wechsel in der Zwischendarstellung kann sich die Struktur eines Programmes teils erheblich verändern, was beachtet werden muss. Einen großen Vorteil bieten die Hochsprachen dennoch, sie enthalten teils große Mengen an impliziten Flow-Facts, die leicht automatisch extrahiert werden können.

Als Eingabe erhält die statische Analyse neben bereits annotierten Flow-Facts den Quellcode oder Binärcode. Dieser wird zunächst in einen CFG überführt (Abbildung 2.4). Auf diesem werden mit Hilfe von Informationen über die Semantik des verwendeten Befehlssatzes Kontrollflussanalysen durchgeführt, um z. B. weitere Flow-Facts gewinnen zu können. Im nächsten Schritt wird durch die Verwendung des abstrakten Architekturmodelles die $\mathrm{WCET}_{\mathrm{est}}$ jedes Basisblockes ermittelt. Im letzten Schritt wird nun der eigentliche WCEP und die zugehörige $\mathrm{WCET}_{\mathrm{est}}$ bestimmt. Dazu werden die in Abschnitt 2.4.1 bereits vorgestellten Verfahren genutzt. Ein Beispiel für ein solches Analyseprogramm stellt a^3 dar, das innerhalb des WCC genutzt wird, und in Abschnitt 3.3 näher beschrieben wird.

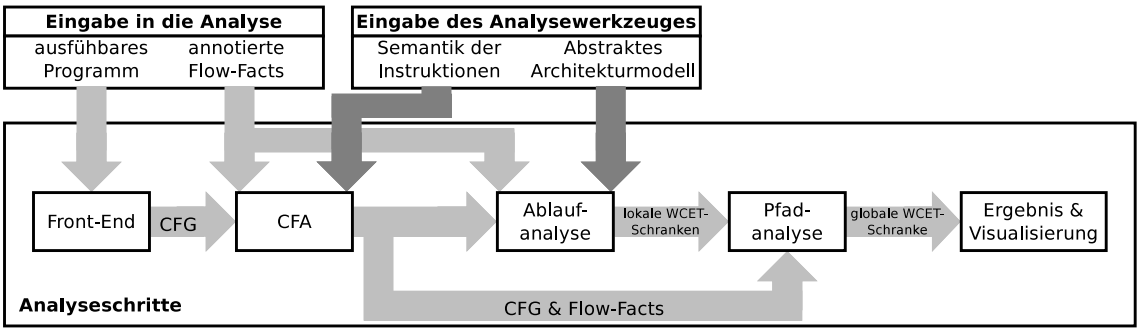


Abbildung 2.4: Schema einer statischen WCET-Analyse nach [WEE+08]

Entscheidend für die Güte des Ergebnisses ist hierbei die Genauigkeit des Architekturmodelles und der Flow-Facts. Man muss bei der Festsetzung von Schranken z. B. von Schleifeniterationen immer zwischen der Exaktheit und Sicherheit abwägen. Zieht man die Exaktheit vor und wählt die Schranken niedrig, so liegt das Ergebnis einer WCET-Analyse vermutlich näher an der tatsächlichen WCET. Man

geht aber das Risiko ein, dass die Approximation keine sichere obere Schranke mehr für die maximale Ausführungszeit darstellt. Favorisiert man jedoch die Sicherheit, kann es leicht zu einer starken Überschätzung bzgl. des $\mathrm{WCET}_{\mathrm{est}}$-Wertes kommen.

Durch die hohe Komplexität der heutigen Architekturen können deren Modelle die eigentliche Funktionalität nur vereinfacht darstellen. Die Verwendung folgender Bestandteile [KP05] macht eine exakte Modellierung sehr schwierig bis unmöglich:

- Vielstufige, superskalare Pipelines mit Forwarding, Stalls und Out-of-Order Execution
- Verwendung von Caches mit verschiedenen Ersetzungsstrategien, deren Inhalt zur Laufzeit nicht immer bekannt ist.
- Verschiedene Latenzen von Caches und Speichern
- Sprungvorhersagen

Die Auswertung der einzelnen Punkte hängt immer vom aktuellen Kontext ab, in dem das Programm ausgeführt wird. Zudem gibt es dynamische Effekte zur Laufzeit [LGZ+09], die die Abschätzung erschweren:

- dynamische Funktionsaufrufe (Lösung durch Benutzereingabe oder Ersetzung durch statische Funktionsaufrufe)
- Schleifen, deren Iterationszahl an Laufzeiteingaben gebunden sind

Interrupts, die auch dynamische Effekte sind, werden i. d. R. nicht berücksichtigt. Bei der Analyse können auch sogenannte **Timing-Anomalien** vorkommen, die z. B. durch spekulatives Vorgehen hervorgerufen werden [WEE+08]. So kann eine spätere Ausführung eines Befehls dazu führen, dass das Programm insgesamt schneller abgearbeitet wird. Daher genügt es in solchen Fällen nicht, immer den lokalen Worst-Case zu betrachten, also die Kante mit den größten Kosten weiterzuverfolgen, sondern alle Verzweigungen müssen weiter verfolgt werden. Sicherheit bietet hier nur der globale Worst-Case, der alle möglichen Ausführungspfade untersucht. Dabei müssen zusätzlich alle möglichen Eingaben und Startzustände berücksichtigt werden. Dadurch ist es notwendig zu abstrahieren, weshalb Präzision verloren geht, was wiederum in einer Überschätzung der WCET resultiert.

2.5.2 Messungsbasierte WCET-Analyse

Bei dieser Variante der WCET-Analyse wird das zu analysierende Programm mehrfach mit verschiedenen Eingaben ausgeführt und die $\mathrm{WCET}_{\mathrm{obs}}$-Werte werden verglichen [WEE+08]. Die höchste $\mathrm{WCET}_{\mathrm{obs}}$ wird als Abschätzung für die WCET benutzt. Diese Variante liefert keine sichere obere Schranke, da niemals alle verschiedenen Programmabläufe getestet werden können und man niemals sicher sein kann, dass auch die Worst-Case-Eingabe und der Worst-Case-Startzustand betrachtet wurden. Gerade die zuvor schon erwähnten Timing-Anomalien machen die Annäherung an den Worst-Case-Startzustand schwierig. Ein Vorteil dieser Variante ist,

dass kein abstraktes Architekturmodell benötigt wird. Zudem liegen nach [WEE+08] die WCET_{obs}-Werte i. d. R. näher an der WCET als die WCET_{est}-Werte der statischen Analysen, vor allem, wenn die Architekturen komplexer sind.

Kombinierte Ansätze benutzen die Vorgehensweise statischer WCET-Analysen (Abbildung 2.4). Sie messen die WCET_{obs} von Bestandteilen des Programmes, anstatt die Ablaufanalyse durchzuführen. Diese Messwerte werden anschließend für die Pfadanalyse benutzt.

2.6 Registerallokation

Die meisten Prozessorarchitekturen verwenden Operationen, die auf Registern arbeiten. Diese sind kleine Speicher innerhalb des Prozessors und werden im Folgenden **physikalische Register** oder einfach nur Register genannt. Ihre Anzahl ist stark begrenzt, während die Anzahl der Variablen innerhalb eines Programmes groß sein kann. Wie die Register verwendet werden, hängt von der **Instruktionsauswahl** ab. In diesem Arbeitschritt werden die Instruktionen des hochsprachennahen Programmes durch passende Instruktionen des Prozessorbefehlssatzes ersetzt. Er kann zum einem nicht mehr benötigte Variablen auf dem Stack ablegen und dafür zusätzliche Lade- und Speicheroperationen einfügen. Auf diese Weise sind immer genügend Register frei. Die andere und hier im Folgenden betrachtete Möglichkeit ist die Erzeugung von virtuellem Programmcode, welcher beliebig viele **virtuelle Register** enthalten kann. Virtuelle (synonym „symbolische“) Register stellen die Variablen des Programmes dar. Sie werden als lebendig bezeichnet, wenn die zugehörige Variable lebendig ist (Definition 2.5). Dazu wird auf dem CFG des Programmes eine Lebendigkeitsanalyse durchgeführt, die für jeden Knoten die aus virtuellen Registern bestehenden Mengen LiveIn und LiveOut bestimmt.

Die zweite Variante der Instruktionsauswahl benötigt den Arbeitsschritt der **Registerallokation**, der die virtuellen Register den wenigen vorhandenen physikalischen Registern zuordnet. Dabei dürfen niemals zwei virtuelle Register dem selben physikalischen Register zugeordnet werden, wenn sie gleichzeitig lebendig sind. Da i. d. R. nie alle virtuellen Register ihre gesamte Lebenszeit in einem physikalischen Register verbringen können, müssen sie zwischenzeitlich in einem der darüberliegenden Speicher abgelegt werden. Dazu werden zusätzliche Lade- und Speicheroperationen eingefügt, die **Spillcode** oder **Spillinstruktionen** genannt werden. Die Entscheidungen solche zusätzlichen Befehle einzufügen werden **Spillentscheidungen** genannt. Zudem heißt die Gesamtheit der Ausführungszeiten der eingefügten Spillinstruktionen **Spillcode-Overhead**. Je höher der verwendete Speicher in der Speicherhierarchie liegt, desto höher sind die Latenzen für die Zugriffe. Ein optimaler Registerallokator sollte nur so viel Spillcode einfügen, wie unbedingt benötigt wird und damit den Spillcode-Overhead und die Ausführungszeit des Programmes minimieren. Besteht aber keine andere Möglichkeit, so sollte er den nötigen Spillco-

de optimal positionieren, damit er möglichst selten ausgeführt wird. Deshalb sollte gerade das Einfügen von Spillcode in Schleifen reduziert werden.

Hennessy und Patterson [HP07] messen der Registerallokation eine ganz besondere Bedeutung unter den Optimierungen bei. Für sie ist die Registerallokation einer der entscheidendsten, wenn nicht sogar der entscheidendste Optimierungsschritt. Gerade hier existiert ein großes Optimierungspotenzial bzgl. der Programmlaufzeit. Zusätzlich macht sie andere Optimierungen erst nutzbar. Die Registerallokation kann dabei an verschiedenen Stellen des Übersetzungsprozesses durchgeführt werden. Sie kann z. B. auf einer Medium-Level-Zwischendarstellung (Abschnitt 3.1) angewendet werden. In diesem Fall dürfen jedoch nicht alle Register genutzt werden, da einige für den Codegenerator reserviert werden müssen. Üblicherweise wird die Registerallokation allerdings auf einer Low-Level-Zwischendarstellung durchgeführt.

Bei der Anwendung anderer Optimierungen muss zudem beachtet werden, das diese die Registerallokation beeinflussen. So können Optimierungen wie das Loop-Unrolling und Function-Inlining dazu führen, dass die Anzahl virtueller Register stark ansteigt. Die erzielten Laufzeitgewinne werden auf diese Weise unter Umständen durch den Spillcode-Overhead wieder zunichte gemacht.

2.6.1 Anforderungen an die Registerallokation

Wie die Registerallokation im Einzelnen abläuft, ist in hohem Maße von der Zielarchitektur abhängig [Muc97]. Einige Eigenschaften des verwendeten Registersatzes müssen speziell modelliert werden. **RISC**-Architekturen (Reduced Instruction Set Computer) z. B. verwenden i. d. R. hauptsächlich Allzweckregister und zusätzlich spezielle FP-Register, während **CISC**-Architekturen (Complex Instruction Set Computer) viele Spezialregister besitzen, die gesonderte Funktionen innehaben.

Calling-Conventions: Bei Funktionsaufrufen muss berücksichtigt werden, welche Calling-Convention vom Compiler angewendet wird. Sie gibt an, wo Übergabeparameter und Rückgabewerte abgelegt werden. Zudem bestimmt sie, welche Register von der aufrufenden und welche von der aufgerufenen Funktion gesichert und später wieder hergestellt werden müssen. Die Register werden dabei in zwei Klassen eingeteilt. Die **Callee-Saved-Register** werden zu Beginn der aufgerufenen Funktion von dieser gesichert und anschließend wieder hergestellt. Bei den **Caller-Saved-Registern** ist die aufrufende Funktion für die Sicherung vor dem Funktionsaufruf zuständig. In diesem Fall darf einem solchen Caller-Saved-Register zu diesem Zeitpunkt kein virtuelles Register zugeordnet sein, da es ansonsten innerhalb der aufgerufenen Funktion überschrieben werden könnte. Diese Register können in der aufgerufenen Funktion unmittelbar benutzt werden. Durch die Calling-Convention sind die Übergabeparameter und Rückgabewerte fest an einzelne Register gebunden. Diese Einschränkung kann mit Hilfe einer **Vorfärbung** umgesetzt werden.

Vorfärbung: Oftmals sind für einige Operationen auch spezielle Register vorgesehen. Bei Funktionsaufrufen müssen z. B. die Calling-Conventions berücksichtigt werden, die Übergabeparameter an genau vorgeschrieben Stellen des Registersatzes ablegen. Einige Architekturen stellen auch Befehle mit impliziten Operanden zur Verfügung, die davon ausgehen, dass der benötigte Wert immer im selben Register liegt. Um eine solche Zuordnung zu ermöglichen, wird ein virtuelles einem physikalischen Register fest zugewiesen, bevor die eigentliche Registerallokation durchgeführt wird. Dieser Vorgang heißt Vorfärbung, weil er im Zusammenhang mit der im nächsten Abschnitt vorgestellten **graphfärbungsbasierten Registerallokation** entstand.

Erweiterte Register: Einige Architekturen unterstützen auch das paarweise Zusammenfassen von Registern, um größere Werte darstellen zu können. Das erweiterte virtuelle Register mit doppelter Größe muss dabei zwei benachbarten physikalischen Registern zugeordnet werden.

2.6.2 Registerallokationsverfahren

Im Bereich der Registerallokationsverfahren unterscheidet man zwischen **lokalen**, **globalen** und **interprozeduralen** Verfahren. Bei lokalen Verfahren werden die Spillentscheidungen nur aufgrund der Betrachtung des aktuellen Basisblocks heuristisch getroffen. Globale Verfahren führen die Zuordnung der Register bzgl. einer ganzen Funktion aus. Die Eigenschaften von Funktionsaufrufen, insbesondere die Einhaltung der Calling-Conventions, werden bei interprozeduralen Verfahren berücksichtigt. Es existiert eine Vielzahl an verschiedenen Verfahren, die das Registerallokationsproblem auf unterschiedlichste Art und Weise auf andere Probleme abbilden oder reduzieren. Viele Ansätze sind heuristisch und können in ungünstigen Fällen einen großen Spillcode-Overhead erzeugen. Andere haben den Anspruch, optimal oder näherungsweise optimal zu sein. Die Lösung mit Hilfe dieser Verfahren kann jedoch sehr viel Zeit einnehmen, weil das Registerallokationsproblem sowohl für globale [Set73] als auch für lokale [FL98] Verfahren NP-vollständig ist. Im Folgenden werden zwei Lösungsansätze kurz erläutert. Andere Ansätze wurden in Abschnitt 1.3 bereits erwähnt.

Graphfärbungsbasierte Registerallokation: Das Standardverfahren im Bereich der Registerallokation ist der graphfärbungsbasierte Ansatz, der erstmals von Chaitin in [Cha81] vorgestellt wurde. Das globale Registerallokationsproblem mit k physikalischen Registern wird dabei auf das k-Graphfärbeproblem reduziert und ist nach Wegener [Weg03] somit für ein $k \geq 3$ NP-vollständig. Deswegen werden im Laufe dieses Verfahrens Heuristiken eingesetzt, die umso besser funktionieren, je mehr physikalische Register zur Verfügung stehen. Die graphfärbungsbasierte Registerallokation zeichnet sich insgesamt vor allem durch ihre Schnelligkeit aus, trotz der durchaus gute Ergebnisse erreicht werden.

In einem ersten Schritt wird ein Interferenzgraph erstellt [App04]. Die Knoten dieses Graphen stehen für die einzelnen virtuellen und physikalischen Register. Zwei Knoten sind miteinander verbunden, wenn die beiden zugehörigen virtuellen Register gleichzeitig lebendig sind und deshalb nicht dem selben physikalischen Register zugeordnet werden dürfen. Ziel ist nun, die Knoten dieses Graphen mit k Farben zu färben, so dass zwei benachbarte Knoten niemals dieselbe Farbe erhalten.

Dazu werden in einem Vereinfachungsschritt zunächst jene Knoten aus dem Interferenzgraphen entfernt, die einen Grad kleiner k und damit keinesfalls mehr als k-1 Nachbarn haben. Diese Knoten sind immer k-färbbar. Alle entfernten Knoten werden auf einem Stack abgelegt. Enthält der Graph anschließend keine Knoten mehr, so ist er k-färbbar, und es muss kein Spillcode eingefügt werden. Dies ist meistens nicht der Fall und der Graph enthält noch Knoten mit Grad $\geq k$. An dieser Stelle kommen aufgrund der Komplexität des k-Graphfärbeproblemes Heuristiken zum Zuge. Es wird nun ein Knoten ausgewählt, der als potenzieller Spillkandidat vorgemerkt wird. Dieser Knoten wird aus dem Graphen entfernt. Der zuvor erwähnte Vereinfachungsschritt und die Auswahl eines Spillkandidaten werden nun so lange abwechselnd durchgeführt, bis der Graph keine Knoten mehr enthält. Zusätzlich kann nach dem Vereinfachungsschritt und vor der Auswahl eines Spillkandidaten ein **Coalescing** (Abschnitt 2.6.3) durchgeführt werden.

Der während der vorhergehenden Schritte erzeugte Stack wird nun wieder schrittweise abgearbeitet, indem die Knoten nach und nach wieder entfernt und nun eingefärbt werden. Nicht als potenzieller Spillkandidat markierte Knoten können immer eingefärbt werden. Bei den markierten muss zunächst überprüft werden, ob die Erzeugung von Spillcode tatsächlich notwendig ist. In diesem Fall wird vor jeder Benutzung und nach jeder Definition des zugehörigen virtuellen Registers Spillcode eingefügt.

Der hier beschriebene Ansatz ist optimistisch und geht zunächst davon aus, dass genügend freie physikalische Register vorliegen, um die virtuellen unterzubringen. Ein pessimistischer Ansatz namens Priority-Based-Graph-Coloring wird von Chow und Hennessy in [CH90] beschrieben. Der Ablauf ist weitgehend identisch, wobei die Erstellung des Interferenzgraphen zu Beginn abweicht. Nicht die Überschneidung der reinen Lebenszeiten führt zum Einfügen von Kanten, sondern die Tatsache, dass zwei Instruktionen im selben Basisblock lebendig sind. Zudem wird angenommen, dass nicht genügend freie Register zur Verfügung stehen. Deshalb werden alle virtuellen Register dem darüberliegenden Speicher zugeordnet und die benötigten nach und nach geladen. Insgesamt soll diese Variante den Vorteil bieten, sensibler auf die Kosten und Vorzüge einzelner Spillentscheidungen reagieren zu können.

ILP-basierte Registerallokation: Das globale Registerallokationsproblem kann auch auf ein ganzzahliges Optimierungsproblem (Abschnitt 2.3) abgebildet werden. Die einzelnen Spillentscheidungen werden durch Entscheidungsvariablen dargestellt. Eine korrekte Zuordnung wird durch das Einfügen von Nebenbedingungen sichergestellt. Auf diese Weise bleibt z. B. gewährleistet, dass niemals zwei virtuelle Re-

gister gleichzeitig dem selben physikalischen Register zugeordnet werden können, wenn diese auch gleichzeitig lebendig sind. Erweiterungen können leicht hinzugefügt werden, z. B. können Vorfärbungen integriert werden, mit deren Hilfe auch Calling-Conventions umgesetzt werden können. Für die Modellierung der Zielfunktion wird ein Gütekriterium benötigt. Möglich wären hierfür Eigenschaften wie die Codegröße, ACET, WCET oder der Energieverbrauch.

Als erstes wurde 1996 ein derartiges Verfahren von Goodwin und Wilken [GW96] vorgestellt. Durch eine entsprechende Platzierung der Spillinstruktionen im Programmcode reduziert es den Spillcode-Overhead auf optimale Weise, so lange gewisse Bedingungen eingehalten werden. Dieses Verfahren bildet die Grundlage des in dieser Diplomarbeit verwendeten Registerallokators. Es wird daher ausführlich im Kapitel 5 dargestellt. Als Zielfunktion wird die $\mathrm{WCET}_{\mathrm{est}}$ des zu übersetzenden Programmes modelliert, die minimiert werden soll. Die Lösung des Optimierungsproblemes ist NP-vollständig. Goodwin und Wilken messen die beobachtete Laufzeit und setzen sie in Verhältnis zur Eingabe. Sie bestimmten auf diese Weise eine Laufzeitkomplexität von $\mathcal{O}(n^3)$ für das Verfahren.

Kong und Wilken [KW98] haben 1998 eine adaptierte Version für irreguläre Architekturen wie den X86 vorgestellt. In der 2002 von Fu und Wilken [FW02] vorgestellten Weiterentwicklung des Verfahrens werden Spillentscheidungen entfernt, die für das Erreichen der Optimalität nicht von Belang sind. Durch das Wegfallen geschieht die Lösung mit Hilfe der verwendeten ILP-Solver weitaus schneller. Die hier beobachtete Laufzeit konnte so auf $\mathcal{O}(n^{2,5})$ reduziert werden. In seiner Dissertation stellte Fu 2007 [Fu07] einen kombinierten Ansatz vor, der Teilprobleme mittels Graphfärbung löst und so schneller arbeitet.

Von Appel und George [AG01] wurde ein zweigeteiltes Verfahren entwickelt. Im ersten Arbeitsschritt wird mit Hilfe eines ILPs bestimmt, welche virtuellen Register zugeordnet werden oder sich im Speicher befinden, und wo Spillcode eingefügt werden muss. Im zweiten Schritt geschieht die eigentliche Zuordnung der virtuellen auf die physikalischen Register. Durch diese Zweiteilung ist die Optimalität nicht mehr gewährleistet, aber die beobachtete Laufzeit beträgt nur $\mathcal{O}(n^{1,3})$.

2.6.3 Optimierungstechniken der Registerallokation

Es existieren eine ganze Reihe verschiedener Optimierungstechniken im Bereich der Registerallokation. Beispielhaft sollen hier drei der bedeutendsten Verfahren näher geschildert werden.

Live-Range-Splitting: Dieses Verfahren teilt die Lebenszeit eines virtuellen Registers in mehrere Abschnitte auf, um Interferenzen zu reduzieren. Auf diese Weise kann die Anzahl der benötigten Register gesenkt werden. Normalerweise wird ein virtuelles Register immer genau einem physikalischen Register zugeordnet. Auch wenn es zwischenzeitlich im Speicher abgelegt wird, muss es anschließend in das selbe physikalische Register zurück geladen werden. Durch Live-Range-Splitting kann

das Register im Laufe seiner Lebenszeit verschiedenen physikalischen Registern zugeordnet werden. Die Aufteilung geschieht durch das Einfügen von Kopierbefehlen und das Umbenennen von Variablen. Durch die Reduzierung der Interferenzen sinkt i. d. R. auch der Umfang des benötigten Spillcodes.

Coalescing/Copy-Elimination: Ziel dieser Optimierung ist das Entfernen von Befehlen, die den Inhalt eines Registers in ein anderes kopieren. Werden zwei virtuelle Register auf das selbe physikalische Register abgebildet und es existiert ein solcher Befehl, so kann dieser entfernt werden. Das Coalescing [Cha81] und die Copy-Elimination [GW96] verfolgen das Ziel, Paare von virtuellen Registern, für die Kopierbefehle existieren, dem selben physikalischen Register zuzuordnen.

In der graphfärbungsbasierten Registerallokation wird das Coalescing hinter dem Vereinfachungsschritt durchgeführt. Gesucht wird ein Kopierbefehl, der den Inhalt des virtuellen Registers s_i in s_j kopiert. Eine weitere Bedingung ist, dass beide Register nicht interferieren oder sie bis zum Ende der Funktion nicht mehr in den Speicher gesichert werden. Die Instruktion, die s_i definiert, wird so geändert, dass sie das Ergebnis in s_j schreibt. Der Kopierbefehl ist nun überflüssig und kann entfernt werden. Durch dieses Verfahren kann jedoch der Grad der einzelnen Knoten erhöht werden, weshalb der Interferenzgraph vielleicht nicht mehr färbbar sein kann. Deshalb wird beim Coalescing differenziert und man spricht von einem sicheren Coalescing, wenn sichergestellt ist, dass der Graph auch im Weiteren färbbar bleibt. Ein sicheres Coalescing stellte Briggs vor [BCT94].

Im ihrem ILP-basierten Ansatz stellen Goodwin und Wilken die Möglichkeit der Copy-Elimination vor. Durch Einfügen zusätzlicher Entscheidungsvariablen und Constraints wird erreicht, dass zwei virtuelle Register dem selben physikalischen Register zugeordnet werden, wenn ein Kopierbefehl existiert und die Möglichkeit dazu besteht. Hat die entsprechende Entscheidungsvariable nach Lösung des ILP den Wert Eins, so kann der Kopierbefehl entfernt werden.

Rematerialization: Das Verfahren der Rematerialization [BCT92] kann eingesetzt werden, um Spillinstruktionen einzusparen. Oftmals kann es günstiger sein, den Inhalt eines virtuellen Registers neu zu berechnen, anstatt ihn über Spillinstruktionen erst im Speicher abzulegen und anschließend wieder zu laden. Voraussetzung ist natürlich, dass die Berechnung ohne weiteren Aufwand wiederholt werden kann.

„Die Worst-Case Execution Time (WCET) zu kennen ermöglicht es eine Hardware-Plattform zu benutzen oder zu entwerfen, die auf den Resssourcenbedarf der Software, wie Speicher oder Taktfrequenz, zugeschnitten ist.“

Heiko Falk, Paul Lokuciejewski und Henrik Theiling
übersetzt aus dem Englischen [FLT06, Seite 121]

3 WCC - WCET-Aware C Compiler

An Compiler für eingebettete Systeme wird die Anforderung gestellt, hoch-optimierten Programmcode zu erzeugen. Dieser soll möglichst kompakt sein, um die Speicherressourcen zu schonen. Ebenso soll das Programm möglichst schnell und energieeffizient ausgeführt werden können. In einigen Fällen ist auch eine Parallelisierung gewünscht. Oftmals ist besonders eine der bedeutendsten Eigenschaften eingebetteter Systeme gefordert, die Echtzeitfähigkeit. Sie setzt explizites Wissen über die WCET voraus.

Als erster Compiler bezieht der WCC (WCET-Aware C Compiler) Ergebnisse einer WCET-Analyse in seine Arbeitsschritte mit ein. Dieser Compiler [FLT06] wird am Lehrstuhl 12[2] der Fakultät Informatik der Technischen Universität Dortmund entwickelt. Er übersetzt ANSI-C-Programme [ISO99] in ausführbare Programme für die Architekturen TC1796 (Kapitel 4) und TC1797 [Inf09]. Die Einbindung der ARM7-Architektur [ARM05] wird derzeit umgesetzt. Zur Ermittlung der WCET_{est}-Daten benutzt der WCC das kommerzielle Analyseprogramm a^3 (Abschnitt 3.3) der AbsInt Angewandte Informatik GmbH. So können diverse Optimierungen angewendet werden, die den WCEP verkürzen.

Am Ende der Übersetzung kann die WCET_{est} der Programme mittels a^3 ermittelt werden. Zum einem kann so sichergestellt werden, dass die Zeitschranken eingehalten werden. Zum anderen können die Produktionskosten drastisch minimiert werden, da das System kleiner dimensioniert werden kann, was z. B. den Prozessor- und Speichertakt angeht. So kann das Hardwaresystem an die verwendete Software speziell angepasst werden.

Zunächst wird der WCC in Abschnitt 3.1 mit dem Aufbau gängiger Compiler verglichen, bevor er im Detail in Abschnitt 3.2 beschrieben wird. Die WCET-Analyse innerhalb des WCC mittels a^3 stellt Abschnitt 3.3 vor.

3.1 Der WCC im Vergleich

Der obere Teil der Abbildung 3.1 stellt das Modell eines Compilers dar [Muc97], der **Zwischendarstellungen (Intermediate Representation)** auf zwei unterschied-

[2] http://ls12-www.cs.tu-dortmund.de/research/activities/wcc/index.html

lichen Ebenen verwendet und nach dem Fließbandprinzip arbeitet. Das Front-End besteht aus einer lexikalischen Analyse, einem Parser, einer semantischen Analyse und einem Codegenerator. Der Programmcode wird in diesem Front-End in eine **MIR** (Medium Level Intermediate Representation) überführt, auf die Optimierungen angewendet werden können. Ein weiterer Codegenerator überführt die MIR in eine **LIR** (Low Level Intermediate Representation). Auf diese können ebenfalls Optimierungen angewendet werden, bevor im letzten Codegenerator abschließend der Assemblercode erzeugt wird.

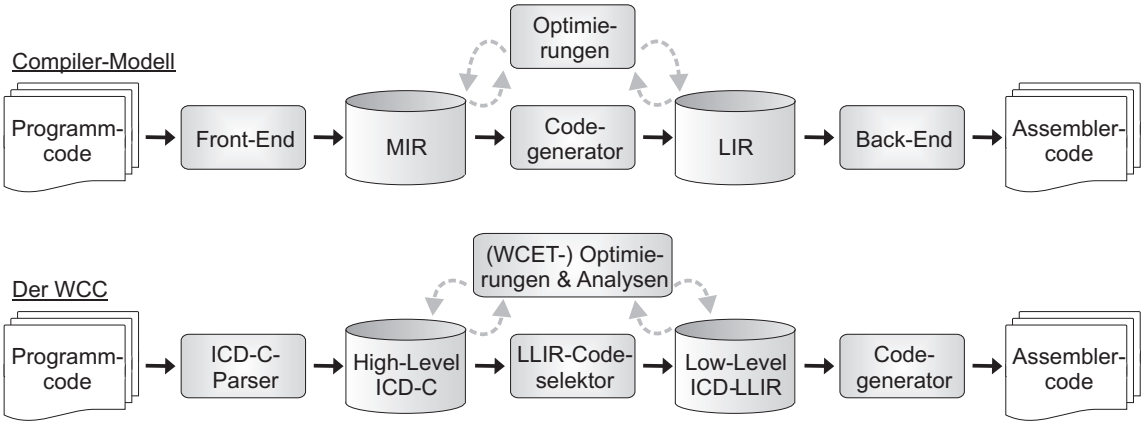


Abbildung 3.1: Compilermodell nach [Muc97] im Vergleich zum WCC nach [FLT06]

Der WCC, im unteren Teil der Abbildung 3.1 vereinfacht dargestellt, verwendet den **ICD-C-Parser**, der die zuvor erwähnten Arbeitsschritte eines Compiler-Front-Ends ausführt. Dadurch überführt er ANSI-C-Programme in die High-Level-Darstellung **ICD-C**. ICD-C [ICD10a] ist ein Compiler-Framework des **ICD**[3] (Informatik Centrum Dortmund), das als Compiler-Front-End eingesetzt werden kann. Auf Grundlage von ICD-C werden Analysen und Optimierungen ausgeführt, insbesondere auch solche, die die WCET gezielt minimieren. Der **LLIR-Codeselektor** erzeugt die Low-Level-Darstellung **ICD-LLIR** (Abschnitt 3.2.1), auf die ebenfalls Analysen und Optimierungen angewendet werden. Der **Codegenerator** überführt diese zum Schluss in Assemblercode.

3.2 Der Aufbau im Detail

Die Abbildung 3.2 zeigt den aus Abbildung 3.1 bereits bekannten Übersetzungsprozess des WCC in detaillierter Form. Der ICD-C-Parser überführt den Quellcode in ICD-C und übernimmt dabei auch die ebenfalls übergebenen Flow-Facts [Sch07]. Diese sind zusätzliche Informationen über die Rekursionstiefen der Funktionen und

[3] http://www.icd.de/es/index.html

die Anzahl der Schleifeniterationen, die den Quellcode erweitern (Abschnitt 2.2). Die resultierende Zwischendarstellung stellt den maschinenunabhängigen C-Code direkt dar. Auf Grundlage von ICD-C können neben Schleifenanalysen auch andere Analysen sowie diverse High-Level-Optimierungen durchgeführt werden. Der Codeselektor verwendet die Methode des Tree-Pattern-Matching, um die ICD-C in die ICD-LLIR zu überführen, die nahe am fertigen Assemblercode und maschinenabhängig ist. Sie bietet die Möglichkeit verschiedenster Analysen und Low-Level-Optimierungen.

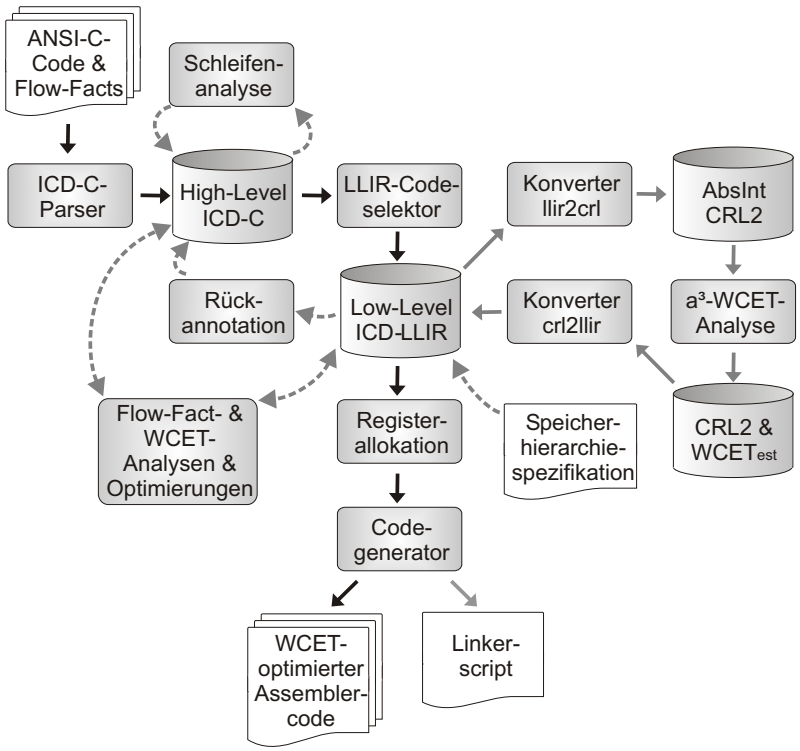


Abbildung 3.2: Aufbau des WCC nach [FLT06]

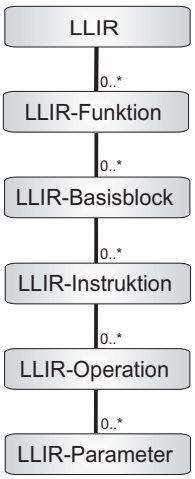
Insbesondere kann von hier auf das Analyseprogramm a^3 (Abschnitt 3.3) zugegriffen werden, das WCET-Analysen zur Verfügung stellt. Die ICD-LLIR wird dafür durch Konverter in die Low-Level-Darstellung **CRL2** umgewandelt. Diese wird nach Beendigung der Analyse wieder rückübersetzt und mit Anmerkungen über die WCET_{est}-Werte und andere Analyseergebnisse versehen. Um die WCET-Analyse überhaupt durchführen zu können, werden die anfangs übergebenen Flow-Facts benötigt. Zusätzlich verbessern die Flow-Facts auch die Güte der WCET-Analyse. Die Ergebnisse werden über die Rückannotation auch den High-Level-Optimierungsschritten zugänglich gemacht. Die ICD-LLIR enthält zunächst virtuelle Register, um Analysen und Optimierungen durchführen zu können. Erst in der Registerallokation werden diese virtuellen durch physikalische Register ersetzt. Als Grundlage für einige Optimierungsschritte wird eine Speicherhierarchiespezifikation benötigt, die festlegt welche Speicher verwendet werden und in welchen Speichern

Daten und Programmcode abgelegt sind. Im letzten Schritt erstellt der Codegenerator aus der ICD-LLIR den Assemblercode, der zusammen mit einem Linkerscript dem Linker übergeben wird.

Das Konzept des Compilers ist flexibel gestaltet und ist nicht auf die Optimierung der WCET beschränkt. Dazu verwendet der WCC sogenannte Objectives, die an alle vorhanden Unterstrukturen innerhalb der Zwischendarstellungen angehängt werden können. Sie enthalten alle Daten, die für weitere Optimierungen nützlich sein können. So können nicht nur Optimierungen bezüglich der WCET ausgeführt werden, sondern z. B. auch Optimierungen, die den Energieverbrauch oder die Codegröße reduzieren.

3.2.1 ICD-LLIR

Die ICD-LLIR (ICD Low Level Intermediate Representation) ist die Low-Level-Zwischendarstellung des ICD [ICD10b]. Sie enthält assemblerartigen Code, der anhand der Struktur aus Abbildung 3.3 angeordnet ist.



Eine LLIR kann mehrere Funktionen enthalten und diese wiederum mehrere Basisblöcke. Basisblöcke bestehen i. d. R. aus mehreren Instruktionen, wobei nur die letzte eine Sprunganweisung sein darf. Die Instruktionen können aus mehreren Operationen zusammengesetzt sein, insbesondere, wenn die verwendete Architektur einen komplexen Befehlssatz verwendet. Die in dieser Diplomarbeit verwendete TC1796-Architektur (Kapitel 4) ist derart auf die ICD-LLIR abgebildet, dass jede Instruktion nur eine Operation umfasst. Operationen können schlussendlich mehrere Parameter besitzen. Diese können Register, Operatoren, konstante Zahlenwerte oder Label sein.

Abbildung 3.3: Aufbau der ICD-LLIR nach [ICD10b]

3.2.2 Registerallokation im WCC

Der WCC verfügt über verschiedene Registerallokationsverfahren, deren Güte und Laufzeit in Abhängigkeit vom zu übersetzenden Programm stark variieren können. Der Standardregisterallokator ist ein einfacher graphfärbungsbasierter Allokator (im weiteren **RAGC** genannt), wie er in Abschnitt 2.6 bereits vorgestellt wurde. Sein größter Vorteil ist die geringe Laufzeit, in der dieser akzeptable Ergebnisse erzeugt. Hier werden jedoch keinerlei Annahmen bzgl. der WCET gemacht. In ungünstigen Fällen kann sich dieses Verfahren sehr negativ auf die WCET auswirken. Zusätzlich verfügt der WCC über einen sehr schnellen und naiven Registerallokator (im

weiteren **RASA** genannt), der vor jeder Benutzung und nach jeder Definition eines Registers Spillcode einfügt. Dieser wird jedoch nicht zur eigentlichen Übersetzung genutzt, sondern nur um ausführbaren Binärcode für die statische Analyse mit a^3 zu erzeugen.

Der WCC hat den Anspruch, ein Compiler zur WCET-Minimierung zu sein und verfügt somit über zwei Registerallokatoren, die sich dieser Zielstellung auf verschiedene Weisen explizit annehmen. Der erste Ansatz [Fal09] (im weiteren **RAGC-WCET** genannt) stellt eine Erweiterung der einfachen Graphfärbung um Heuristiken zur WCET-Minimierung dar. Müssen im Laufe der Allokation Spillentscheidungen getroffen werden, so versucht dieser Allokator die bzgl. der WCET am günstigsten erscheinenden Entscheidungen zu treffen. Dazu muss der WCEP bekannt sein. Damit dieser gefunden werden kann, muss der Code ausführbar sein. Deshalb werden alle virtuellen Register im Speicher abgelegt, analog zum RASA. Dadurch bleibt der Code ausführbar, und die Spillentscheidungen, die den WCEP minimieren sollen, können getroffen werden. Die Instabilität des WCEP macht es jedoch erforderlich, die Analyse während des Vorgangs mehrfach zu erneuern. Aus praktischen Gesichtspunkten kann dies nicht nach jeder Spillentscheidung geschehen. Daher wird dieser Vorgang erst nach der Abarbeitung ganzer Basisblöcke durchgeführt. Der zweite WCET-optimierende Ansatz stellt die Grundlage des in dieser Diplomarbeit erweiterten Registerallokators (im weiteren **RAILP-WCET** genannt) dar und wird in Kapitel 5 ausführlich beschrieben. Beide Verfahren sind deutlich zeitintensiver als RAGC.

Zuletzt verfügt der WCC noch über einen codegrößenoptimierenden Registerallokator (im weiteren **RAILP-CS** genannt), der den selben ILP-basierten Ansatz wie in Abschnitt 5.1 beschrieben verwendet. Dieser Allokator versucht so wenige Spill-Instruktionen wie irgend möglich einzufügen. Er trifft dabei keinerlei Annahmen bzgl. der WCET und erzeugt dabei erstaunlich gute Ergebnisse die WCET betreffend. Für komplexere Benchmarks benötigt dieser Allokator deutlich mehr Zeit als RAGC, aber i. d. R. weniger als die WCET-basierten Ansätze.

3.3 WCET-Analyse im WCC

Die WCET-Analyse innerhalb des WCC geschieht mit Hilfe des Analysewerkzeuges a^3 (englische Aussprache: „a cube“) der AbsInt Angewandte Informatik GmbH [Abs10]. Dieses zeichnet sich vor allem durch seine präzisen Ergebnisse und umfangreichen Analysen aus. Da es ein statisches Analysewerkzeug ist, wird das zu analysierende Programm hier niemals ausgeführt. Das Ergebnis stellt im Falle einer sorgfältigen Konfiguration eine sichere obere Schranke für harte Echtzeitanwendungen dar. Die $\mathrm{WCET}_{\mathrm{est}}$ liegt i. d. R. nur knapp über der WCET.

Als Eingabe wird eine lauffähige Binärdatei benötigt (Abbildung 3.4), die zuerst innerhalb des Konverters von einem Disassembler in die von a^3 verwendete Low-

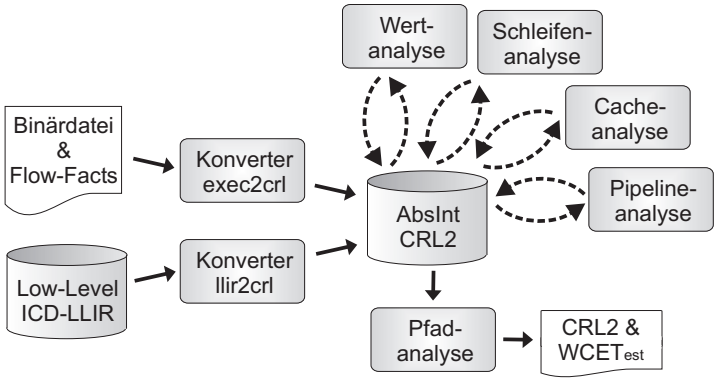


Abbildung 3.4: WCET-Analyse mit a^3

Level-Zwischendarstellung CRL2 (Control Flow Representation Language Version 2) rückübersetzt wird. Zunächst wird dabei der CFG wiederhergestellt. Dieser wird dann in CRL2 übersetzt. Der WCC selber ist auf eine andere Weise an a^3 angebunden. Der Konverter llir2crl überführt die ICD-LLIR des WCC in CRL2. Für die ICD-LLIR muss dafür zunächst eine korrekte Registerallokation durchgeführt werden, da diese zuvor nur virtuelle Register besitzt. Im CRL2-Format hinterlegt a^3 alle Zwischenergebnisse der einzelnen Analyseschritte. Zusätzlich zum eigentlichen Programm werden Flow-Facts benötigt. Diese müssen im Vorhinein manuell gesammelt werden, da nur einfache Fälle von a^3 automatisch analysiert werden können. Die nun folgenden Analyseschritte bauen schrittweise aufeinander auf:

Wertanalyse: Die Wertebereiche aller möglichen Registerinhalte zu allen Ausführungszeitpunkten werden mittels abstrakter Interpretation ermittelt. Auch wenn die genauen Registerinhalte nicht immer exakt bestimmt werden können, kann a^3 durch die Wertebereiche präzisere Abschätzungen treffen. Insbesondere sei hier angemerkt, dass bei diesem Analyseschritt, sowie allen folgenden, jeweils alle Ausführungskontexte separat betrachtet und deren Werte hinterlegt werden können, um eine möglichst präzise $\mathrm{WCET}_{\mathrm{est}}$ bestimmen zu können.

Schleifenanalyse: Hier wird eine obere und untere Schranke für die Anzahl der Schleifeniterationen bestimmt. Die Möglichkeit dazu besteht jedoch nur in einfachen Fällen. Alle komplizierteren Fälle hingegen müssen der Analyse über Flow-Facts mitgeteilt werden. Analog werden hier auch Rekursionen behandelt. Eine präzise Wertanalyse stellt die Voraussetzung für eine präzise Schleifenanalyse dar. Diese ist wiederum oftmals besonders entscheidend für eine präzise $\mathrm{WCET}_{\mathrm{est}}$, vor allem für Programme, die viel Rechenzeit in Schleifen verbringen.

Cacheanalyse: Nach einem durch den Benutzer definierten Cachemodell werden alle Speicherzugriffe als Cache-Hit oder Cache-Miss eingestuft. Der Benutzer wählt dabei aus, wie die Speicherzugriffe gewertet werden, ob z. B. alle als Cache-Miss angesehen werden oder unsichere immer als Cache-Hit. Diese Wahl hat einen großen

Einfluss auf die Güte der WCET_{est} und kann im schlechtesten Falle sogar dazu führen, dass diese unter der WCET liegt, oder sehr weit darüber.

Pipelineanalyse: Anhand eines präzisen Pipelinemodells wird der Programmablauf für alle möglichen Kontexte analysiert. Dies alles geschieht in Abhängigkeit des aktuellen Zustands der Pipeline und aller Ressourcen, sowie der Ergebnisse der vorhergehenden Analysen. Als Ergebnis erhält jeder Basisblock eine Liste mit seinen WCET_{est}-Werten für alle möglichen Ausführungskontexte. Je nach Konfiguration der Cacheanalyse kann es hier zu Verzweigungen kommen. Der Benutzer hat die Auswahl, ob die WCET-Analyse hier global oder lokal ausgewertet wird. Bei der lokalen Auswertung wird immer der Zweig fortgeführt, der momentan am teuersten erscheint. Dies kann wiederum das Ergebnis nachteilig beeinflussen. Eine globale Analyse hingegen kann sehr viel aufwändiger bzgl. der Rechenzeit und des Speicherverbrauchs sein.

Pfadanalyse: Auf Grundlage der ermittelten WCET_{est}-Werte und der Iterationswerte aller Basisblöcke wird mit Hilfe der IPET-Methode (Abschnitt 2.4.1) der WCEP und damit die WCET_{est} des Gesamtprogrammes bestimmt. Dabei werden alle möglichen Ausführungspfade berücksichtigt, was die Genauigkeit des Analyseergebnisses verbessert.

Am Ende der Pfadanalyse stehen die WCET_{est} -Werte für das gesamte Programm und aller einzelnen Funktionen und Basisblöcke zur Verfügung. Bei den letzteren beiden liegen insbesondere die Werte für alle Ausführungskontexte vor. Zusätzlich liegen die Werte für die Iterationshäufigkeiten vor. Innerhalb der Analyse werden auch Pfade erkannt, die niemals ausgeführt werden können. Sämtliche Informationen werden nach Abschluss über einen WCET-Handler an die ICD-LLIR angehängt. Durch die präzise Modellierung innerhalb von a^3 und eine genaue Annotation der Flow-Facts können die Ergebnisse für Arbeitsschritte wie die Registerallokation und Optimierungen im WCC unmittelbar eingesetzt werden.

„Der TriCore vereint das beste dreier Welten: die Echtzeitfähigkeit eines Mikrocontrollers, die Rechenleistung eines DSPs und die besten Leistungsmerkmale und Preisvorteile einer RISC-Architektur . . . “

Infineon Technologies AG
übersetzt aus dem Englischen [Inf02, Seite 7]

4 Zielarchitektur

Diese Diplomarbeit verwendet den TC1796 (TC für TriCore™) der Firma Infineon als Zielarchitektur. Er wurde speziell für die Belange eingebetteter Echtzeitsysteme entworfen [Inf02]. Der Name TriCore™ soll verdeutlichen, dass er sowohl die Eigenschaften eines Mikrocontrollers, eines DSPs (Digitaler Signalprozessor) sowie eines RISC-Prozessors umfasst. Durch diesen Funktionsumfang eignet er sich für den Einsatz in der Automatisierungstechnik, Netzwerktechnik, mobilen Kommunikation, in Computerperipheriegeräten und besonders im Fahrzeugbau.

Zuerst soll in Abschnitt 4.1 die Architektur des TC1796 vorgestellt werden. Anschließend gehen die Abschnitte 4.2 und 4.3 genauer auf die Bestandteile der Architektur wie der Prozessorpipeline und des Registersatzes ein. Der Befehlssatz des TC1796 wird in Abschnitt 4.4 beschrieben. Im Abschnitt 4.5 wird am Ende des Kapitels das Pipelineverhalten der Architektur beschrieben.

4.1 Architektur

Der TC1796 (Abbildung 4.1) ist ein 32-Bit-Microcontroller, der dem Prinzip einer Harvard-Architektur folgt [Inf02]. An der TriCore-CPU (Central Processing Unit) ist die FPU (Floating Point Unit) direkt angeschlossen, um Fließkommaoperationen mit einfacher Genauigkeit durchführen zu können. Die CPU kann auf den SPB (System Peripheral Bus) zugreifen, um mit anderen Geräten zu kommunizieren. Besonders an dieser Architektur ist, dass Daten und Programme nicht nur in verschiedenen Speichern abgelegt sind, sondern auch über verschiedene Busse transferiert werden. Das PMI (Program Memory Interface) stellt einen 16 KB Befehls-Cache sowie einen 48 KB Scratchpad-Speicher [4] (SPM von Scratchpad Memory) für Programmcode zur Verfügung und ermöglicht der CPU den Zugriff auf den PLMB (Program Local Memory Bus).

Über diesen können unter anderem Befehle aus der PMU (Program Memory Unit) bzw. Daten aus der DMU (Data Memory Unit) gelesen oder geschrieben werden.

[4] Scratchpad-Speicher sind kleine schnelle Speicher, auf die der Prozessor schnell zugreifen kann. Sie können Daten oder Programme enthalten. Zudem ist ihr Inhalt bekannt, da dieser Übersetzungszeit festgelegt wird. Dadurch lassen sich präzisere Laufzeitvorhersagen treffen als bei der Verwendung von Caches.

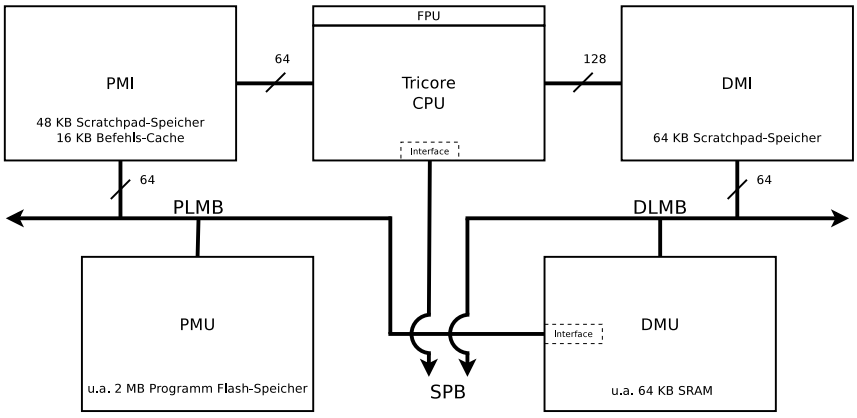


Abbildung 4.1: Vereinfachte Darstellung der TC1796-Architektur nach [Inf07]

Sowohl PMU als auch DMU verfügen dabei über verschiedene Einzelspeicher für Programme bzw. Daten. Analog zum PMI bietet das DMI (Data Memory Interface) die Möglichkeit, auf den 64 KB Daten-SPM und den DLMB (Data Local Memory Bus) zuzugreifen. Hierbei kann über den DLMB unter anderem auf die DMU und den SPB zugegriffen werden. Der Adressraum hat eine Gesamtgröße von 4 GB und ist in 16 gleich große Segmente unterteilt, die verschiedenen Geräten und Aufgaben zugeteilt sind.

Der TC1796 hat den Anspruch realzeitfähig zu sein, wofür die Verwendung eines Programm-SPM und einer schnellen Interruptbehandlung stehen. Sie sind ebenso typische Eigenschaften eines Microcontrollers wie die Unterstützung von Bitoperationen. Zudem verfügt der TC1796 über die besondere Eigenschaft schneller Kontextwechsel. Ein kombinierter **Befehlssatz** (Abschnitt 4.4) aus 16- und 32-Bit-Befehlen ermöglicht Optimierungen bezüglich der Programmgröße. DSP-typische Eigenschaften des TC1796 sind die Unterstützung von Sättigungsarithmetik, MAC-Befehlen (Multiply Accumulate), Zero-Overhead-Looping und vieler spezieller Adressierungsmodi. Merkmale einer RISC-Architektur erkennt man bei Betrachtung der **Prozessorpipeline** (Abschnitt 4.2), sowie des Register- und Befehlssatzes. Die Prozessorpipeline besteht aus drei einzelnen Pipelines mit nur vier Stufen. Der Registersatz besteht hauptsächlich aus 32 Allzweckregistern und wenigen Spezialregistern. Auf komplexe Befehle, die komplexe Pipelinestufen voraussetzen, wird im Befehlssatz verzichtet. Alle Befehle arbeiten ausschließlich auf Registern, wobei für Speicherzugriffe Lade- und Speicherbefehle verwendet werden.

4.2 Prozessorpipeline

Der TC1796 verfügt über eine RISC-typische superskalare Prozessorpipeline (Abbildung 4.2). Sie besteht aus drei einzelnen Pipelines mit den vier Pipelinestufen **Fetch**, **Decode**, **Execute** und **Writeback** [Inf04]. In der ersten Stufe werden die

Befehle aus dem Speicher geladen. Die Decode-Stufe untersucht die Befehle auf die verwendeten Operanden und mögliche Konflikte hin. In der Execute-Stufe werden z. B. Berechnungen ausgeführt. Die Writeback-Stufe verändert den Inhalt des Registersatzes.

Der linke Zweig in Abbildung 4.2 stellt die **I-Pipeline** (auch IP im weiteren genannt, von Integer-Pipeline) dar, die arithmetische und logische Operationen auf Datenregistern ausführt. Sie besteht aus den zuvor genannten vier Stufen. Innerhalb der Execute-Stufe werden die Berechnungen in der ALU (Arithmetic Logic Unit) durchgeführt. Zusätzlich werden hier auch MAC-Befehle und Bitoperationen ausgeführt. Zu beachten ist hierbei, dass MAC- und Multiplikationsbefehle die Execute-Stufe mehrfach durchlaufen können und sie dort einen bis drei Zyklen verweilen. Hierdurch steigt die Latenz für die Verfügbarkeit des Ergebnisses (Tabelle 4.1). Nachfolgende IP-Befehle müssen entsprechend in der Fetch-Stufe warten, bis die Execute-Stufe wieder frei wird. Die I-Pipeline leitet zudem die FP-Befehle (Floating Point) zur FPU weiter, diese verhalten sich dabei wie Mehrzyklen-IP-Befehle.

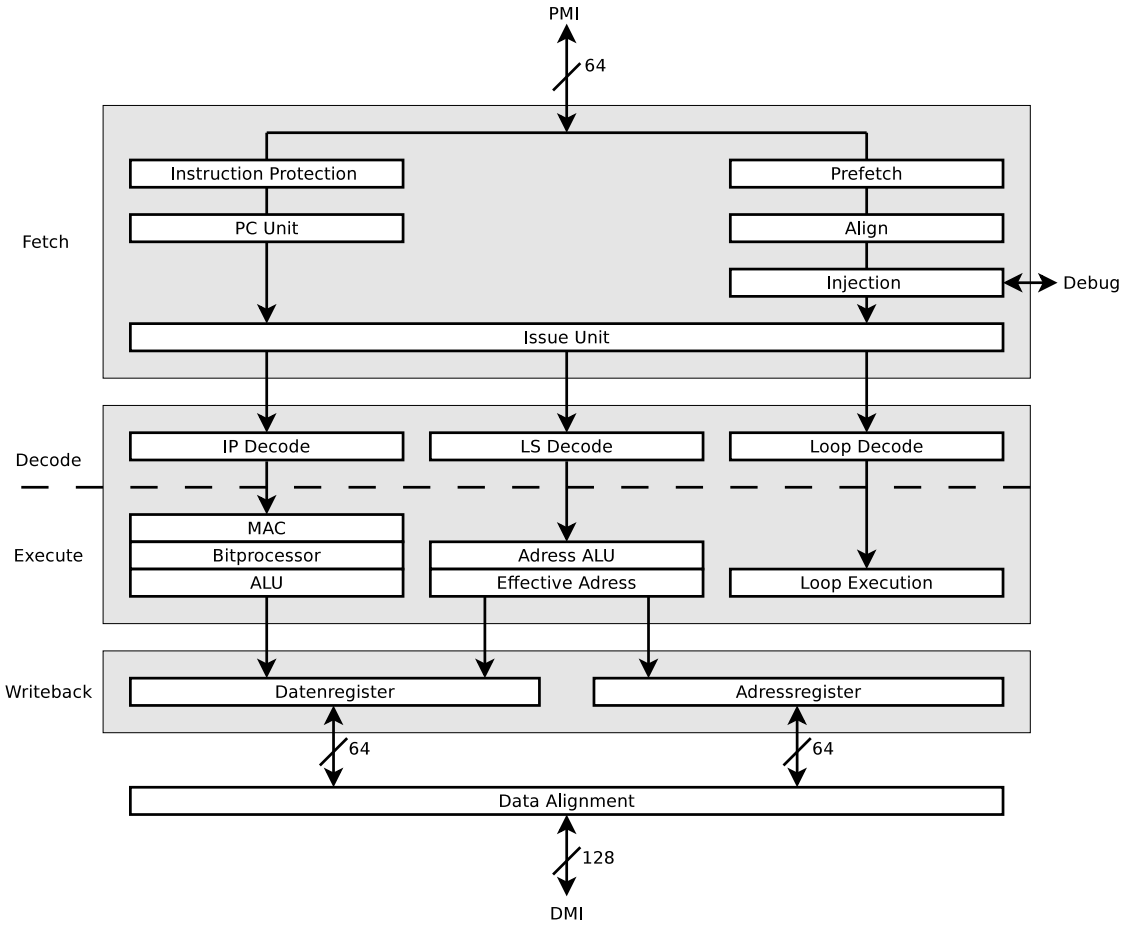


Abbildung 4.2: Prozessorpipeline des TC1796 vereinfacht dargestellt nach [Inf07]

Der mittlere Zweig in Abbildung 4.2 stellt die **LS-Pipeline** (Load/Store) dar, in der LD- und ST-Befehle (Load und Store), arithmetische Operationen auf Adressregistern und unbedingte Sprünge ausgeführt werden. Bedingte Sprünge sowie eine Reihe anderer Instruktionen benötigen die Integer- und LS-Pipeline gleichzeitig für ihre Ausführung (Tabelle 4.2 zweite Spalte). Die im rechten Zweig dargestellte **Loop-Pipeline** verkörpert keine eigenständige Pipeline, da die Loop-Befehle in der LS-Pipeline ausgeführt werden. Dazu verfügt die Architektur über einen Loop-Cache mit zwei Einträgen, in denen die Adressen der Loop-Befehle der innersten beiden Schleifen stehen. Somit kann ein Zero-Overhead-Looping realisiert werden, indem der Schleifenzähler innerhalb der LS-Pipeline automatisch hochgezählt wird und der Sprung ebenfalls automatisch geschieht, sobald die Adresse der Loop-Befehle erreicht wird.

Die beiden Hauptpipelines sind gut balanciert und für die Ausführung einfacher Befehle ausgelegt. Jede Stufe benötigt dabei einen Zyklus. Ein IP-Befehl und ein nachfolgender LS-Befehl können parallel die Fetch-Stufe verlassen und verarbeitet werden, sofern kein Konflikt vorliegt. Steht der LS-Befehl zuerst, ist keine Parallelverarbeitung möglich. Loop-Befehle können immer parallel ausgeführt werden. Die sogenannten DP-Befehle (Dual Pipeline) verwenden beide Pipelines und verhindern die parallele Ausführung von IP- und LS-Befehlen. Optimal für die Auslastung der Pipeline wäre ein Programm, welches abwechselnd aus unabhängigen IP- und LS-Befehlen bestünde.

Befindet sich eine zu ladende Adresse im Cache, so liegt das Ergebnis des Ladevorgangs am Ende des Zyklus vor. Es entstehen Wartezyklen, wenn jedoch z. B. auf die DMU zugegriffen werden muss. LD-Befehle sind blockierend und würden deshalb die ganze Pipeline so lange anhalten, bis das Ergebnis vorliegt. Der TC1796 gewährt jedoch LD-Befehlen Vorrang vor ST-Befehlen. Letzte werden aufgeschoben, so lange kein Konflikt vorherrscht. Um Stalls zu vermeiden, unterstützt die Architektur auch **Forwarding**.

- Es gibt ein Execute-Execute-Forwarding in der I-Pipeline. Dieses ist jedoch eingeschränkt bei MAC-Befehlen. Dort darf das Ergebnis nur als Akkumulator-Eingabe benutzt werden und nicht als Faktor.

- Wird in ein Datenregister geladen, so liegt das Ergebnis den nachfolgenden Befehlen durch Forwarding zu Beginn der Execute-Stufe vor. Bei einem Adressregister kommt es jedoch zu einem Stall (Abschnitt 4.5).

- Zusätzlich gibt es noch ein Forwarding von der IP-ALU zur LS-Pipeline, so dass ST-Befehle parallel ausgeführt werden können und das Ergebnis am Ende der Writeback-Stufe weggespeichert werden kann.

4.3 Registersatz

Der Registersatz (Abbildung 4.3) des TC1796 besteht aus 32 GPRs (General Purpose Register) mit jeweils 32 Bit [Inf02]. Von den 16 Adressregistern A[0] bis A[15] sind vier Register für die Verwendung durch das Betriebssystem reserviert. Die Register A[10] bzw. A[11] enthalten den Stack-Pointer bzw. die Rücksprungadresse. Die 16 Datenregister D[0] bis D[15] sind hingegen frei verwendbar. Eine spezielle Funktion haben die Register A[15] und D[15]. Sie dienen als implizite Operanden für einige Befehle. Zusätzlich zu diesen 32 Registern sind drei weitere vorhanden. Das Register PC (Program Counter) verwaltet die Adresse des aktuell auszuführenden Befehles. Die Register PCXI und PSW enthalten weitere Programmstatusinformationen.

Der Registersatz ist unterteilt in den oberen und unteren Kontext. Die Register A[10] bis A[15], D[8] bis D[15], PCXI sowie PSW gehören dem oberen Kontext an. Zum unteren Kontext gehören die Register A[2] bis A[7], D[0] bis D[7], sowie PC. Die Register A[0], A[1], A[8], A[9] sind keinem Kontext zugeordnet. Der obere Kontext wird bei einem Funktionsaufruf oder der Einleitung einer Interruptroutine automatisch gesichert. Für schnelle Kontextwechsel optimiert, kann der TC1796 die Sicherung des oberen Kontextes in zwei bis vier Zyklen durchführen. Beim Rücksprung oder dem Verlassen der Interruptroutine kann der obere Kontext ebenso schnell wieder hergestellt werden. Der untere Kontext kann separat über Befehle gesichert und geladen werden.

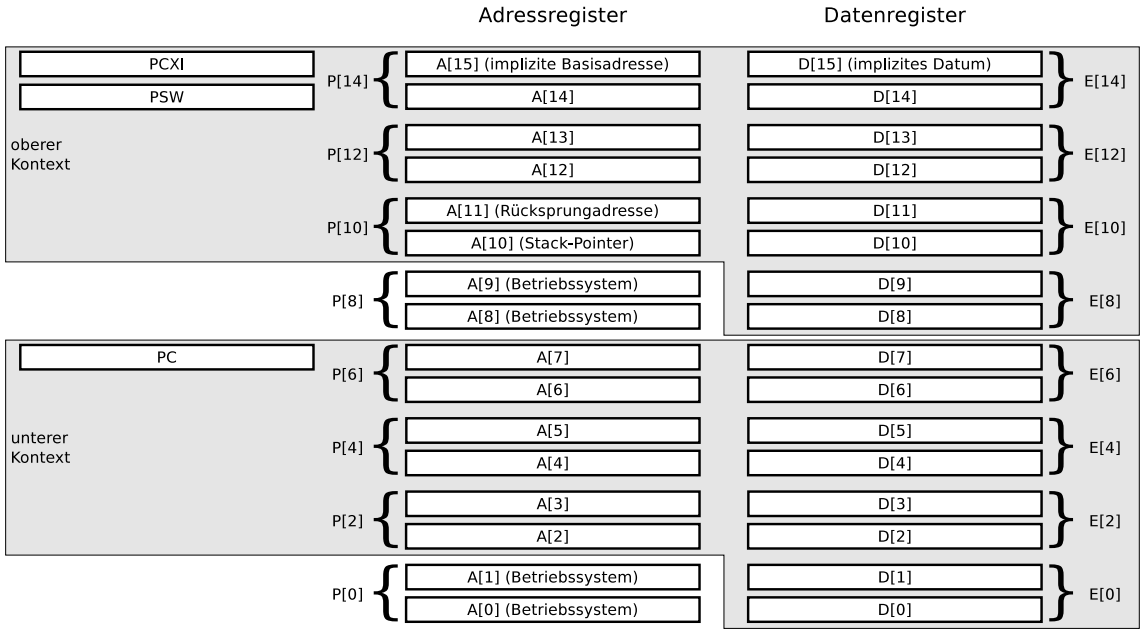


Abbildung 4.3: Der Registersatz des TC1796 nach [Inf08, Inf02]

Da einige Befehle Register mit mehr als 32 Bit benötigen, können Register paarweise als erweiterte Register mit 64 Bit verwendet werden [Inf08]. Erweiterte Daten-

register werden mit E benannt, erweiterte Adressregister mit P. Dabei kann nur ein Register mit einer geraden Nummer und das entsprechende Register mit der nachfolgenden ungeraden Nummer zusammengefasst werden. So kann z. B. das Register D[2] mit seinem Nachbarregister D[3] zusammengefasst werden.

4.4 Befehlssatz

Da der TC1796 eine RISC-Architektur ist, besitzt er keine komplexen Befehle, und alle Befehle arbeiten auf Registern [Inf02]. Die Befehle des TC1796 sind i. d. R. Zwei- oder Drei-Adress-Befehle. Einige Befehle verwenden das Register A[15] als implizite Adresse oder das Register D[15] als implizites Datum.

Alle Befehle, die in der I-Pipeline ausgeführt werden, sind in Tabelle 4.1 aufgelistet. Zusätzlich vermerkt sind **Latenz** und **Wiederholungsrate**. Latenz steht in diesem Fall für die Anzahl der Zyklen, die nach dem Eintritt in die Execute-Stufe vergehen, bis das Ergebnis vorliegt. Die Wiederholungsrate [Inf07] gibt an, nach wie vielen Zyklen ein Befehl mit dem selben Befehlscode im Anschluss frühestens ausgeführt werden kann, wenn kein Konflikt vorliegt. Zwischen Befehlen mit unterschiedlichem Befehlscode muss immer eine Pause liegen, die um Eins niedriger ist als die Latenz. Befehle mit einem Zyklus Latenz müssen demnach trivialerweise in zwei Zyklen hintereinander ausgeführt werden, egal ob ein Konflikt vorliegt oder nicht. MAC- und Multiplikationsbefehle verbringen mehrere Zyklen in der Execute-Stufe. Wenn nun zwei aufeinander folgende Befehle den selben Befehlscode haben und konfliktfrei sind, können sie parallel in der Execute-Stufe verarbeitet werden. Bei MAC-Befehlen ist dies der Fall, wenn für den zweiten Befehl das Ergebnis des ersten Befehls nicht als Faktor benutzt wird. Die Sternchen innerhalb der Tabellen deuten an, dass es sich hier um eine Zusammenfassung mehrerer Befehle handelt, deren Bezeichnungen den selben Anfang haben.

Alle in der ersten Spalte der Tabelle 4.2 aufgelisteten Befehle können in der LS-Pipeline ausgeführt werden, ebenso die Loop-Befehle. Die LD- bzw. ST-Befehle dienen dazu, Inhalte mit unterschiedlicher Bit-Länge aus dem Speicher in die Register bzw. aus den Registern in den Speicher zu transferieren. So gibt es entsprechende Befehle für 8, 16, 32 und 64 Bit. In der zweiten Spalte sind unter DP-Befehle alle Befehle aufgetragen, die beide Pipelines belegen. Zuletzt sind in der vierten Spalte der Vollständigkeit halber noch einmal alle FP-Befehle aufgelistet.

In Tabelle 4.3 ist die jeweilige Ausführungsdauer für alle Befehle der I- und der LS-Pipeline zeilenweise aufgetragen. Die Dauer hängt dabei teilweise vom Vorgängerbefehl ab, der in den Spalten unterschieden wird. Die mit einem Sternchen versehenen Einträge zeigen an, wie groß die Ausführungsdauer wird, wenn keine Datenabhängigkeit vorliegt. Bei Mehrzyklenbefehlen kommt die Bedingung hinzu, dass der Vorgängerbefehl den selben Befehlscode haben muss und das Ergebnis des Vorgängerbefehls nicht als Faktor benutzt werden darf. In allen anderen Fällen ist die

Tabelle 4.1: IP-Befehle mit Latenz und Wiederholungsrate [Inf03a, Inf07]

IP-Befehle	Latenz	Wiederholungsrate
ABS*	1	1
ADD* (ohne ADD_A*, ADDIH_A, ADDSC*)	1	1
AND*	1	1
CADD*	1	1
CL*	1	1
CMOV*	1	1
CSUB*	1	1
DEXTR	1	1
EQ* (ohne EQ_A, EQZ_A)	1	1
EXTR*	1	1
GE* (ohne GE_A)	1	1
INS*	1	1
LT* (ohne LT_A)	1	1
MADD*	2-3	1-2
MAX*	1	1
MIN*	1	1
MOV* (ohne MOV_AA, MOVH_A)	1	1
MSUB*	2-3	1-2
MUL*	2-3	1-2
NAND*	1	1
NE	1	1
NOR*	1	1
NOT	1	1
OR*	1	1
RSUB*	1	1
SAT*	1	1
SEL*	1	1
SH*	1	1
SUB* (ohne SUB_A*)	1	1
X*	1	1

Ausführungsdauer gleich dem unmarkierten Wert. Der Wert Null gibt an, dass der Befehl parallel zum Vorgängerbefehl ausgeführt werden kann. Das heißt, dass ein unabhängiger LD-Befehl z. B. parallel zu einem beliebigen IP-Befehl ausgeführt werden kann. Besteht jedoch eine Datenabhängigkeit, muss er immer separat ausgeführt werden und hat Kosten von einem Zyklus. Zu beachten ist hier jedoch, dass eine eigene Datenabhängigkeit zu einem Mehrzyklen-Vorgängerbefehl für den jeweiligen Befehl irrelevant sein kann, wie z. B. bei einem ST-Befehl. Sie kann jedoch bewirken, dass seine Nachfolger nicht mehr parallel zu diesem Vorgängerbefehl ausgeführt werden können.

Tabelle 4.2: LS-, Loop-, DP- und FPU-Befehle [Inf03a, Inf07]

LS-Befehle	DP-Befehle	Loop-Befehle	FP-Befehle
ADD_A*	ADDIH_A	LOOP*	ADD_F
EQ_A	ADDSC*		CMP_F
EQZ_A	BISR*		DIV_F
GE_A	CACHEA*		FTOI
J_16	CALL*		FTOQ31
J_32	DEBUG*		FTOU
LD*	DISABLE		ITOF
LEA	DSYNC		MADD_F
LT_A	DVADJ		MSUB_F
MOV_AA	DVINIT*		MUL_F
MOVH_A	DVSTEP*		Q31TOF
NE_A	ENABLE		QSEED
NEZ_A	IMASK		SUB_F
NOP*	ISYNC		UPDFL
ST*	J* (ohne J_16, J_32)		UTOF
SUB_A*	MFCR		
	MTCR		
	RET*		
	RFE*		
	RS*		
	SVLCX		
	SWAP		
	SYSCALL		
	T*		

Tabelle 4.3: Ausführungsdauer in Abhängigkeit vom Vorgängerbefehl [Inf07]

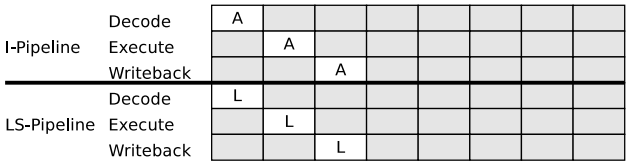
Vorgänger Befehl	IP ohne MAC,MUL	LD und abh. LS-Befehle	ST und unabh. LS-Befehle	MAC 16x16 16x32	MAC 32x32	MUL
IP ohne MAC, MUL	1	1	1	1	1	1
LD und abh. LS-Befehle	0* 1	1	1	0* 1	0* 1	0* 1
ST und unabh. LS-Befehle	0	1	1	0	0	0
MAC 16x16, 16x32	2	2	2	1* 2	2	2
MAC 32x32	3	3	3	3	2* 3	3
MUL	1/2/3	1/2/3	1/2/3	1/2/3	1/2/3	1/1/2* 1/2/3

4.5 Pipelineverhalten

Dieser Abschnitt soll noch einmal detailliert die Funktionsweise der Pipeline verdeutlichen, indem einige Spezialfälle behandelt werden. Insbesondere sollen Gründe [Inf04] für Konflikte und die daraus resultierenden Stalls erläutert werden. Ein genaueres Wissen über die Details der Befehlsausführung innerhalb der Pipeline ist notwendig, um die Kosten von eingefügten Spillinstruktionen bestimmen zu können.

Wie schon in Abschnitt 4.2 erwähnt, können IP- und LS-Befehle parallel verarbeitet werden, wenn kein Konflikt vorliegt. Eine solche Situation ist in Abbildung 4.4 dargestellt. Ebenso können Mehrzyklen-IP-Befehle wie MAC- und Multiplikationsbefehle parallel zu unabhängigen LS-Befehlen ausgeführt werden. Im zweiten Beispiel werden zwei LD-Befehle parallel zu einem MAC-Befehl ausgeführt, weil keine Abhängigkeiten bestehen. Der zweite MAC-Befehl kann direkt die Decode-Stufe verlassen, da er den selben Befehlscode hat wie der Erste und keine Abhängigkeiten bestehen. Der anschließende Additionsbefehl hingegen muss warten, bis die Execute-Stufe wieder frei wird.

```
ADD     d0,d1,d2        ;A
LD      d1,[a0]0        ;L
```




```
MADD.Q  d0,d0,d1,d2     ;M1
LD      d9,[a0]0        ;L1
LD      d10,[a1]0       ;L2
MADD.Q  d3,d3,d4,d5     ;M2
ADD     d6,d7,d8        ;A
```

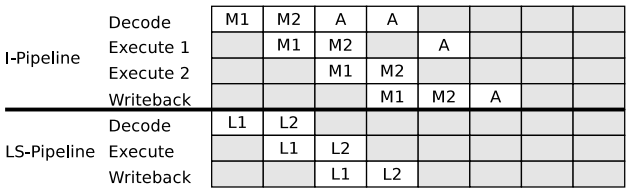


Abbildung 4.4: Parallele Ausführung von LD-Befehlen nach [Inf04]

Im Folgenden werden sechs Fälle angeführt, in denen Stalls entstehen. Stalls betreffen alle nachfolgenden Befehle und halten diese in der Decode-Stufe an [Inf04]. Werden ein IP-Befehl und ein LS-Befehl parallel ausgeführt, so werden beide angehalten, wenn einer durch einen Konflikt betroffen ist. Ein einzelner, nachfolgender IP-Befehl wird bei einem Stall ebenfalls angehalten. Ausnahme ist ein einzelner, nachfolgender LS-Befehl, dieser wird bei einem Stall der I-Pipeline weiter ausgeführt. Im Weiteren wird der Begriff der Datenabhängigkeit in drei Unterfälle unterschieden, dem WAW (Write-After-Write), RAW (Read-After-Write) und WAR (Write-After-Read).

1. WAW-Konflikt zwischen einem IP- und Ladebefehl: In diesem Fall definieren ein IP-Befehl und ein LD-Befehl das selbe Register. Hier liegt ein WAW-Konflikt vor und damit ein struktureller Hazard. Nur eine Pipeline kann in

der Writeback-Stufe auf das Register zugreifen, daher wird der LD-Befehl in der Decode-Stufe für einen Zyklus angehalten.

```
ADD     d0,d1,d2        ;A
LD      d0,[a0]0        ;L
```

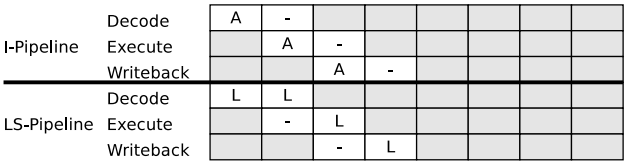


Abbildung 4.5: Fall 1: WAW-Konflikt nach [Inf04]

2. RAW-Konflikt zwischen einem Lade- und LS-Befehl: Wird ein Adressregister geladen, so entsteht ein zusätzlicher Wartezyklus bis das Ergebnis genutzt werden kann [Inf03b]. Der Grund liegt darin, dass die effektive Adresse in der Decode-Stufe schneller berechnet wird als in der Execute-Stufe. Durch Forwarding in der Execute-Stufe können Datenregister hingegen direkt im nächsten Zyklus verwendet werden. In diesem Fall kann zwischen beiden Befehlen noch ein IP-Befehl stehen.

```
LD.A    a5,[a0]0        ;L
ADD.A   a4,a5,a6        ;AA
```

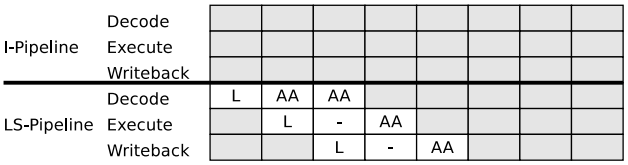


Abbildung 4.6: Fall 2: Delayzyklus nach einem LD.A nach [Inf03b]

3. Ein Speicher- und Ladebefehl greifen auf die selbe Adresse zu: In diesem Fall kommt es zu einem Stall, wenn ein ST-Befehl und ein nachfolgender LD-Befehl auf die selbe Speicheradresse zugreifen [Inf04]. Der Grund liegt in einem strukturellen Hazard. Während der ST-Befehl in der Writeback-Stufe die Speicherstelle beschreiben will, versucht der LD-Befehl zeitgleich in der Execute-Stufe auf die Stelle zuzugreifen.

```
ST      [a0]0,d1        ;S
LD      d2,[a0]0        ;L
```

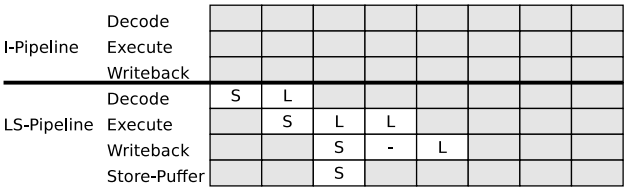


Abbildung 4.7: Fall 3: Konflikt zwischen ST- und LD-Befehl nach [Inf04]

Wie schon zuvor erwähnt, würde ein LD-Befehl Vorrang vor einem ST-Befehl bekommen, um Wartezyklen zu vermeiden. In diesem Fall erkennt die Hardware jedoch den Konflikt und lässt dem ST-Befehl den Vortritt. Der LD-Befehl wird erst weiter ausgeführt, wenn der Speichervorgang abgeschlossen ist. Auch in diesem Fall kann ein IP-Befehl zwischen den beiden Befehlen stehen.

4. Ein LS-Befehl folgt auf einen Funktionsaufruf oder Rücksprungbefehl: Folgt auf einen Funktionsaufruf direkt ein LS-Befehl, der am Anfang der aufgerufenen Funktion steht, so wird dieser unter Umständen angehalten [Inf03b]. Ein Funktionsaufruf entspricht einem Multi-Zyklen-ST-Befehl, der den oberen Kontext sichert, in Kombination mit einem Sprung. Liegt der LS-Befehl z. B. im Cache, so ist dieser nämlich ausführbereit, obwohl der obere Kontext noch nicht vollständig gesichert wurde. Zwischen einem Funktionsaufruf und LS-Befehl kann ein IP-Befehl stehen, der wiederum direkt ausgeführt werden kann.
5. Ein Speicherbefehl steht vor einem Rücksprungbefehl: Folgt auf einen ST-Befehl ein Rücksprungbefehl, so muss dieser einen Zyklus warten, bis der ST-Befehl komplett abgearbeitet wurde [Inf03b]. Ein Rücksprungbefehl entspricht dabei einem Multi-Zyklen-LD-Befehl, der den oberen Kontext lädt, kombiniert mit einem Sprung. Bevor die Wiederherstellung des oberen Kontextes vorgenommen werden kann, muss der Store-Puffer leer sein, ähnlich wie im dritten Fall.
6. Konflikt zwischen einem Mehrzyklen-IP- und einem Speicher- oder Ladebefehl: Der sechste Fall verhält sich analog zum ersten Fall, mit den Unterschied, dass es sich hier um Mehrzyklenbefehle handelt. Nachfolgend sind zwei Situationen dargestellt. Zuerst wird ein abhängiger LD-Befehl angehalten, weil ein WAW-Konflikt vorliegt und nicht beide Pipelines gleichzeitig auf das selbe Register zugreifen dürfen. Der MAC-Befehl wird zuerst ausgeführt, bevor der LD-Befehl in die Writeback-Stufe gelangt. In der zweiten Situation liegt ein RAW mit einem ST-Befehl vor. Anzumerken ist, dass der ST-Befehl einen Zyklus weniger angehalten wird, da durch Forwarding das Ergebnis in der Writeback-Stufe vorliegt. In beiden Fällen kann der Nachfolgebefehl jedoch nicht parallel zum MAC-Befehl ausgeführt werden, obwohl dieser keine Datenabhängigkeit aufweist.

```
MADD.Q  d0,d0,d1,d2     ;M
LD      d0,[a0]0        ;L1
LD      d3,[a1]0        ;L2
```

I-Pipeline	Decode	M							
	Execute 1		M						
	Execute 2			M					
	Writeback				M				
LS-Pipeline	Decode	L1	L1	L1	L2				
	Execute		-	-	L1	L2			
	Writeback			-	-	L1	L2		

```
MADD.Q  d0,d0,d1,d2     ;M
ST      [a0]0,d0        ;S
LD      d3,[a1]0        ;L
```

I-Pipeline	Decode	M							
	Execute 1		M						
	Execute 2			M					
	Writeback				M				
LS-Pipeline	Decode	S	S	L					
	Execute		-	S	L				
	Writeback			-	S	L			

Abbildung 4.8: Fall 6: WAW-Konflikt nach einem MAC-befehl nach [Inf04, Inf03b]

„Der ORA-Ansatz startet mit der Analyse des Kontrollflussgraphen der Funktion, um ein 0-1 ILP zu erzeugen, das alle Entscheidungen einer Registerallokation enthält, die für eine optimale Allokation getroffen werden müssen.“

David W. Goodwin und Kent D. Wilken
übersetzt aus dem Englischen [GW96, Seite 931]

5 ILP-basierte Registerallokation im WCC

Der WCET-optimierende Registerallokator des WCC wurde von Schmoll [Sch08] im Rahmen seiner Diplomarbeit am Lehrstuhl 12 entwickelt. Das Verfahren arbeitet auf Basis der Low-Level-Darstellung ICD-LLIR. Dazu wird das Registerallokationsproblem auf ein ILP abgebildet, welches mit dem ILP-Solver CPLEX gelöst wird. Abschließend werden die Ergebnisse anhand der Variablenwerte der Lösung wieder in die ICD-LLIR übertragen. Die Registerallokation innerhalb der Funktionen geschieht global, dabei wird jede Funktion durch ein einzelnes ILP modelliert. Insgesamt wird ein interprozeduraler Ansatz verfolgt, weil die Calling-Conventions für die Funktionsaufrufe zusätzlich modelliert werden.

Der Aufbau des modellierten ILPs lässt sich in zwei Teile gliedern. Der erste Teil besteht aus einer Menge von Nebenbedingungen, die eine korrekte Registerallokation erzeugen. Das Verfahren orientiert sich an dem ILP-basierten Ansatz von **Goodwin und Wilken** [GW96] und wird im folgenden Abschnitt detailliert erläutert. Der zweite Teil des ILPs stellt die Zielfunktion dar. Auf Basis der von **Suhendra et al.** in [SMRC05] benutzten Methode wird der WCEP als Zielfunktion modelliert, dessen Ausführungszeit minimiert werden soll. Die Erläuterung dieser Methode folgt im Abschnitt 5.2. Im Anschluss wird der gesamte Ablauf der Registerallkation noch einmal in Abschnitt 5.3 zusammengefasst. Die Schwachpunkte dieser Implementierung, die Schmoll in seiner Diplomarbeit auch erwähnt, werden in Abschnitt 5.4 angeführt. Im Abschnitt 5.5 werden die Lösungsansätze, die innerhalb dieser Diplomarbeit umgesetzt werden, vorgestellt. Dieser Abschnitt bildet im Besonderen eine Überleitung zu den folgenden Kapiteln.

5.1 Modellierung der Registerallokation nach Goodwin und Wilken

Im ORA-Ansatz (Optimal Register Allocation) von Goodwin und Wilken [GW96] werden alle Zustände und Aktionen in Form von Entscheidungsvariablen ausgedrückt. Wird einer dieser Variablen bei der Lösung des ILPs der Wert Eins zugewie-

sen, ist dieser Zustand aktiv oder wird die Aktion ausgeführt. Der Ansatz führt eine optimale Registerallokation und Spillcodeerzeugung durch, wenn die Spillinstruktionen derart eingefügt werden, dass die zugehörige WCET minimiert wird. Voraussetzung hierfür ist, dass der Einfluss, den eingefügte Spillinstruktionen auf die WCET haben, nur durch deren Art und ihre Ausführungshäufigkeit beeinflusst wird. Die Ausführungshäufigkeit wird im WCC mit Hilfe des Analyseprogrammes a^3 bestimmt und richtet sich dabei nach dem Basisblock, in welchen die Spillinstruktion eingefügt wird.

Durch die ILP-Modellierung lassen sich Erweiterungen einfach bewerkstelligen. So zeigen Goodwin und Wilken stufenweise den modularen Aufbau ihres Verfahrens. Insbesondere erläutern sie die Integration der Optimierungsverfahren Copy-Elimination und Rematerialization (Abschnitt 2.6.3), welche Schmoll nicht in seine Implementierung übernommen hat. Das Verfahren des Live-Range-Splittings ist hingegen implizit durch die Art der Modellierung enthalten. Unter Voraussetzung der folgenden Annahmen liefert das ILP eine optimale Lösung für die Registerallokation. Die Anforderungen für die Rematerialization werden hier nicht erwähnt.

- Die Registerallokation ändert nicht die Reihenfolge der bereits vorhandenen Instruktionen.
- Es werden nur Spillinstruktion in Form von Lade- und Speicheroperationen einfügt.
- Es werden nur überflüssige Kopierinstruktionen entfernt, bei denen beide virtuellen Register durch die Lösung des ILPs demselben physikalischen Register zugeordnet wurden.
- Alle Pfade im Kontrollflussgraphen können durchlaufen werden.
- Für jede Spillinstruktion sind die Kosten einer Ausführung und ihre Ausführungshäufigkeit bekannt.
- Die Ergebnisse der im Vorfeld angewandten WCET-Analyse müssen den Kirchhoffschen Gesetzen folgen. Ein Basisblock mit einem Vorgänger z. B. darf demnach maximal genauso oft ausgeführt werden wie dieser.

5.1.1 Registerallokation

In einem ersten Schritt werden zunächst für jedes virtuelle Register s genau k verschiedene **VRGs** (Virtual Register Graph) mit Hilfe der Lebendigkeitsanalyse erzeugt, wobei k die Anzahl der physikalischen Register ist. Der VRG ist eine spezielle Form des CFGs, der mit Hilfe eines **IGs** (Instruction Graph) erzeugt wird. Beide sind wie folgt definiert:

Definition 5.1 (IG - Instruction Graph) *Der IG zu einem virtuellen Register s enthält den Teilgraph des CFG, in dem das virtuelle Register s lebendig ist. Jede Instruktion wird durch einen Knoten repräsentiert, der nur einen Vorgänger und*

Nachfolger besitzen kann. Des Weiteren wird jeder Basisblock durch einen Start- und Endknoten dargestellt. An diesen Knoten können Verzweigungen und Zusammenführungen stattfinden. Alle Knoten sind entsprechend ihrer Reihenfolge im Programmablauf über gerichtete Kanten miteinander verbunden. Der Wurzelknoten entspricht dem Knoten, an dem s das erste Mal definiert wird.

Definition 5.2 (VRG - Virtual Register Graph) *Für jedes virtuelle Register s existiert zu jedem zur Verfügung stehenden physikalischen Register p ein VRG. Dieser wird schrittweise aus dem IG durch die Anwendung der Transformationsregeln erzeugt, die in den Abbildungen 5.2, 5.3 und 5.7 (a) und (b) dargestellt sind. Er enthält alle Knoten und Kanten, die auch der zugehörige IG enthält. Hinter den Kanten, an denen Spillentscheidungen getroffen werden müssen, werden ein neuer Knoten und eine neue Kante gemäß der Transformationsregeln in Abbildung 5.7 (a) und (b) eingefügt. Alle Kanten bzw. jene Knoten, die ein virtuelles Register benutzen, werden mit Entscheidungsvariablen annotiert, die modellieren, ob s dem Register p zugeordnet bzw. s in den Speicher verschoben wird. Zusätzlich enthält der VRG die Nebenbedingungen, die durch die Anwendung der Transformationsregeln entstehen und die Korrektheit der Allokation gewährleisten.*

Ein Beispiel für einen CFG und die zugehörigen IGs der virtuellen Register findet man in Abbildung 5.1. Für jedes virtuelle Register s und seinen zugehörigen IG werden die VRGs durch die Transformierungsschritte aus den Abbildungen 5.2 und 5.3 erzeugt. Die Kanten des VRGs mit den dazugehörigen Entscheidungsvariablen werden erzeugt, indem der jeweilige IG traversiert wird. Jene Entscheidungsvariablen geben an, ob das jeweilige virtuelle Register s dem physikalischen Register p zu diesem Zeitpunkt zugeordnet ist. Nach der Lösung des ILPs müssen die Register entsprechend der Variablenbelegung in die ICD-LLIR übertragen werden. Ist s dem Register p nicht mehr zugeordnet, muss es im Speicher abgelegt werden. Des Weiteren bleibt hier zu beachten, dass durch die implizite Modellierung des Live-Range-Splitting ein virtuelles Register im Verlauf seiner Lebenszeit verschiedenen physikalischen Registern zugeordnet werden kann. Steht ein physikalisches Register zum Zeitpunkt nicht zur Verfügung, so ist der entsprechende VRG leer und die zugehörigen Entscheidungsvariablen werden nicht erzeugt.

An einem Knoten, in dem das virtuelle Register s definiert wird, erhält die ausgehende Kante eine Entscheidungsvariable von der Gestalt $x^{def}_{s\to p}$ (Abbildung 5.2 a). Sie gibt an, ob s nach seiner Definition dem physikalischen Register p bis zur nächsten Benutzung zugeordnet bleibt. Nach einer Benutzung von s muss entschieden werden, ob s weiterhin p zugeordnet bleibt. Anderenfalls wird s über Spillinstruktionen in den Speicher verschoben, und p ist frei für eine andere Zuordnungen. Hierzu werden zwei Entscheidungsvariablen $x^{cont}_{s\to p}$ bzw. $x^{end}_{s\to p}$ (Abbildung 5.2 b) eingefügt, welche modellieren, ob s beibehalten bzw. weggespeichert wird. Zusätzlich wird am Knoten eine Nebenbedingung der Form $x^{pre} = x^{end}_{s\to p} + x^{cont}_{s\to p}$ eingefügt, um sicherzustellen, dass genau eine der beiden Variablen auch wirklich auf den Wert Eins gesetzt wird.

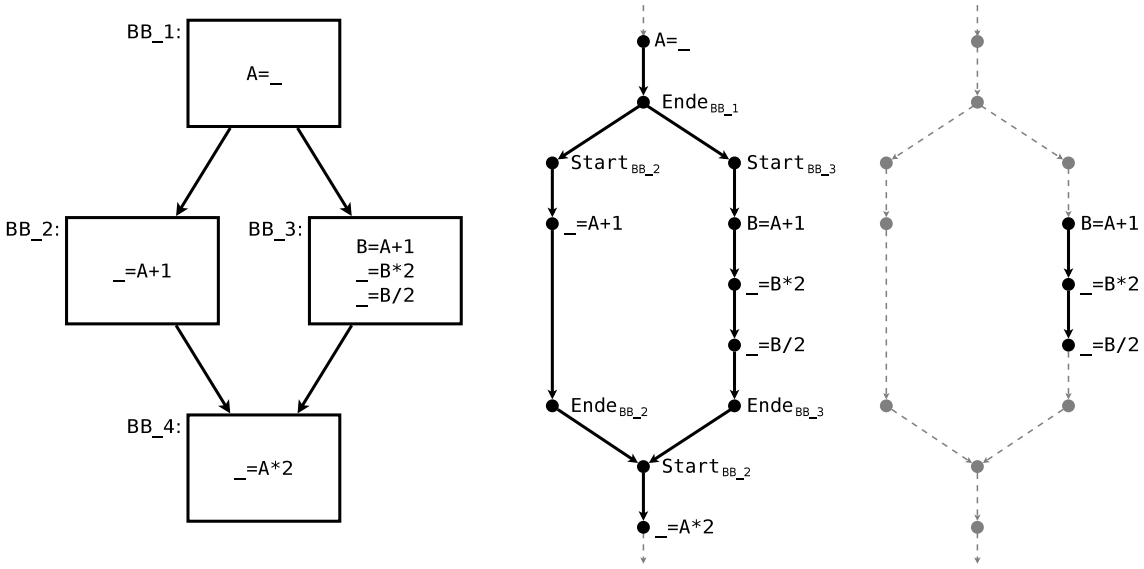


Abbildung 5.1: Konstruktion von IGs aus einem CFG nach [GW96]

Die Variable x^{pre} steht hier und im Folgenden als Platzhalter für die an der eingehenden Kante annotierten Entscheidungsvariablen. Wird s das letzte mal von einer Instruktion benutzt, was gleichzeitig dessen Lebenszeitende entspricht, muss am zugehörigen Knoten die Entscheidungsvariable gelten, mit der die eingehende Kante annotiert ist (Abbildung 5.2 c).

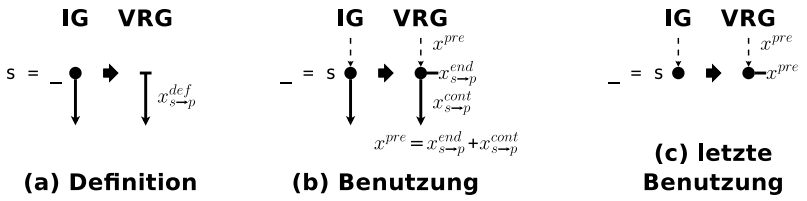


Abbildung 5.2: Transformation der Knoten eines IGs in die eines VRGs nach [GW96]

An den ausgehenden Kanten von Knoten, die s weder definieren noch benutzen, gilt die Entscheidungsvariable, mit der die eingehende Kante annotiert ist (Abbildung 5.3 a). Nach Verzweigungen gilt an beiden ausgehenden Kanten die Entscheidungsvariable, die an der eingehenden Kante annotiert ist (Abbildung 5.3 b). Nach einer Zusammenführung erhält die ausgehende Kante eine der Entscheidungsvariablen der beiden eingehenden Kanten (Abbildung 5.3 c). Zusätzlich wird eine Nebenbedingung eingefügt, die sicherstellt, dass beide Entscheidungsvariablen der eingehenden Kanten den selben Wert zugewiesen bekommen.

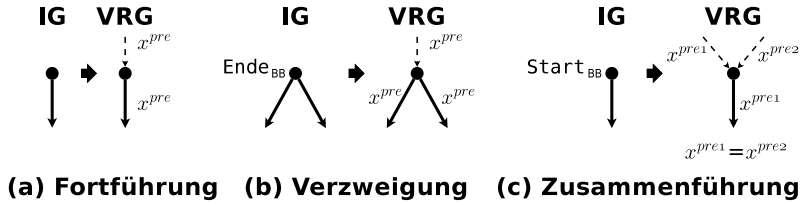


Abbildung 5.3: Kontrollflusstransformation von einem IG in einen VRG nach [GW96]

Wendet man diese Transformationsschritte nun auf die IGs (Abbildung 5.1 b und c) an, so erhält man die zugehörigen VRGs (Abbildung 5.4 a und b). Zu beachten ist hier, dass das Register p als Platzhalter für alle möglichen physikalischen Register steht, da jedes dieser Register einen separaten VRG besitzt. Zusätzlich sind auch unterhalb der Graphen die Nebenbedingungen aufgeführt. Jene Stellen, die ihre Erzeugung verursacht haben, sind zudem durch Nummern gekennzeichnet.

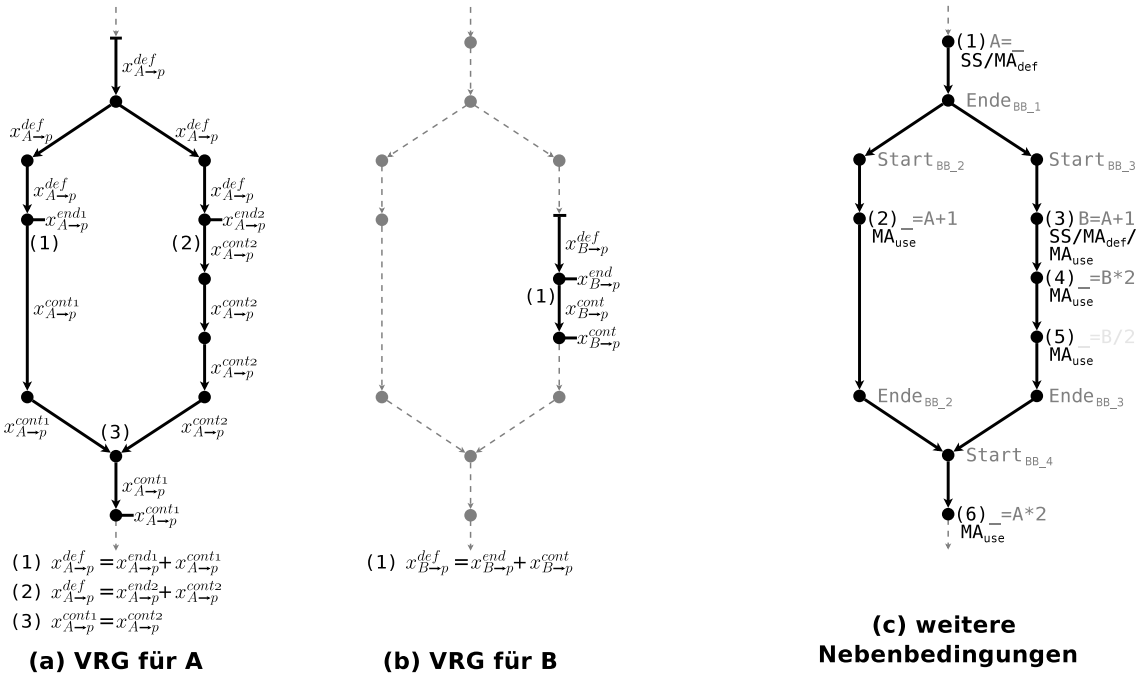


Abbildung 5.4: Erzeugte VRGs und Stellen weiterer Nebenbedingungen nach [GW96]

In Abbildung 5.4 (c) befindet sich ein allgemeiner IG, der die Lebenszeit beider virtuellen Register abdeckt und in diesem Fall identisch mit dem des virtuellen Registers A ist. In diesem sind die Stellen gekennzeichnet, die für das Einfügen weiterer Nebenbedingungen verantwortlich sind. Es gibt drei Arten von Nebenbedingungen, die sicherstellen, dass das gelöste ILP eine korrekte Registerallokation repräsentiert.

- Zuerst muss mittels einer **Single-Symbolic-Bedingung** (SS) sichergestellt werden, dass jedem physikalischen höchstens ein virtuelles Register gleichzeitig zugeordnet werden kann. An Kanten, die von einem definierenden Knoten ausgehen, muss deshalb für jedes physikalische Register p die Summe aller $x^{def}_{s\to p}$ und $x^{cont}_{s\to p}$ kleiner oder gleich Eins sein.
- Ebenfalls muss nach einer Definition durch eine **Must-Allocate$_{\text{def}}$-Bedingung** (MA_{def}) gewährleistet sein, dass das definierte virtuelle exakt einem physikalischen Register zugeordnet ist. Die Summe aller Entscheidungsvariablen $x^{def}_{s\to p}$, die das entsprechende virtuelle Register betreffen, muss an solchen Kanten deshalb genau Eins sein.
- Zuletzt muss vor der Benutzung eines virtuellen Registers über eine **Must-Allocate$_{\text{use}}$-Bedingung** (MA_{use}) sichergestellt sein, dass dieses mindestens einem physikalischen Register zugeordnet ist. Dazu genügt es an dieser Stelle zu überprüfen, ob im Anschluss mindestens eine der Entscheidungsvariablen $x^{cont}_{s\to p}$ und $x^{end}_{s\to p}$, die das entsprechende virtuelle Register betreffen, den Wert Eins annimmt. Die Summe muss dementsprechend größer oder gleich Eins sein.

Anhand des bisher betrachteten Beispiels ergeben sich für die verschiedenen Stellen (Abbildung 5.4 c) folgende zusätzliche Nebenbedingungen:

	Single-Symbolic	Must-Allocate$_{\text{def}}$	Must-Allocate$_{\text{use}}$
(1)	$\forall p : x^{def}_{A\to p} \leq 1$	$\sum_p x^{def}_{A\to p} = 1$	
(2)			$\sum_p x^{end}_{A\to p} + x^{cont}_{A\to p} \geq 1$
(3)	$\forall p : x^{def}_{B\to p} + x^{cont}_{A\to p} \leq 1$	$\sum_p x^{def}_{B\to p} = 1$	$\sum_p x^{end}_{A\to p} + x^{cont}_{A\to p} \geq 1$
(4)			$\sum_p x^{end}_{B\to p} + x^{cont}_{B\to p} \geq 1$
(5)			$\sum_p x^{cont}_{B\to p} \geq 1$
(6)			$\sum_p x^{cont}_{A\to p} \geq 1$

Abbildung 5.5: Weitere Nebenbedingungen nach [GW96]

Bei der Benutzung eines erweiterten Registers werden für dieses zunächst VRGs in Zweierpaaren für jede mögliche Paarung von physikalischen Registern erzeugt. Die Kanten nach einer Definition erhalten in beiden VRGs dieselbe Entscheidungsvariable $x^{def}_{s\to p}$. Für die Entscheidungsvariablen, die nach einer Benutzung folgen, muss über eine zusätzliche Nebenbedingung sichergestellt sein, dass diese immer den gleichen Wert zugewiesen bekommen.

5.1.2 Spillcodeerzeugung

Mit der im vorherigen Abschnitt vorgestellten Modellierung lässt sich das Problem der globalen Registerallokation unter der Einschränkung lösen, dass immer ausreichend viele physikalische Register zur Verfügung stehen. Anderenfalls muss Spillcode erzeugt werden. Goodwin und Wilken weisen in [GW96] nach, dass mit Hilfe der vorgestellten Modellierung und unter Voraussetzung der am Anfang genannten Bedingungen (Abschnitt 5.1) der nötige Spillcode optimal eingefügt werden kann.

Die Abbildungen 5.6 (a) und (b) zeigen beispielhaft Stellen, an denen Spillentscheidungen eingefügt werden. Diese sind durch Rauten dargestellt. Für Spillinstruktionen, die virtuelle Register im Speicher ablegen, genügt es demnach, nur zwei Arten von Kanten zu betrachten, an denen diese eingefügt werden können. Diese Kanten heißen **Spill-Speicher-Kanten**. Entweder gehen sie direkt von einer Instruktion aus, die das virtuelle Register definiert oder benutzt. Sie können aber auch die Kanten sein, die auf nachfolgende Basisblöcke zeigen, wenn der aktuelle Basisblock mehrere Nachfolger besitzt.

Analog verhält es sich für Spillinstruktionen, die virtuelle Register laden und **Spill-Lade-Kanten** heißen. Sie können entweder unmittelbar vor der benutzenden Instruktion oder dem aktuellen Basisblock eingefügt werden, wenn dieser mehrere Vorgänger hat. Zu beachten ist, dass diese Spillinstruktionen virtuelle Register definieren, und deshalb an ihren ausgehenden Kanten noch Single-Symbolic-Bedingungen eingefügt werden müssen.

Aus Gründen der Kompatibilität zur bestehenden Struktur des WCC nimmt Schmoll an dieser Stelle eine Abänderung vor. Er verschiebt die Spill-Speicher-Kanten bzw. Spill-Lade-Kanten, die eigentlich auf den Anfang eines Basisblocks zeigen, in den Basisblock dahinter bzw. davor (Abbildung 5.6 c und d). Bei den Spill-Lade-Kanten muss hier insbesondere beachtet werden, dass ein Sprungbefehl am Ende eines Basisblockes die letzte Instruktion bleibt und die Spill-Lade-Kante entsprechend davor eingefügt wird.

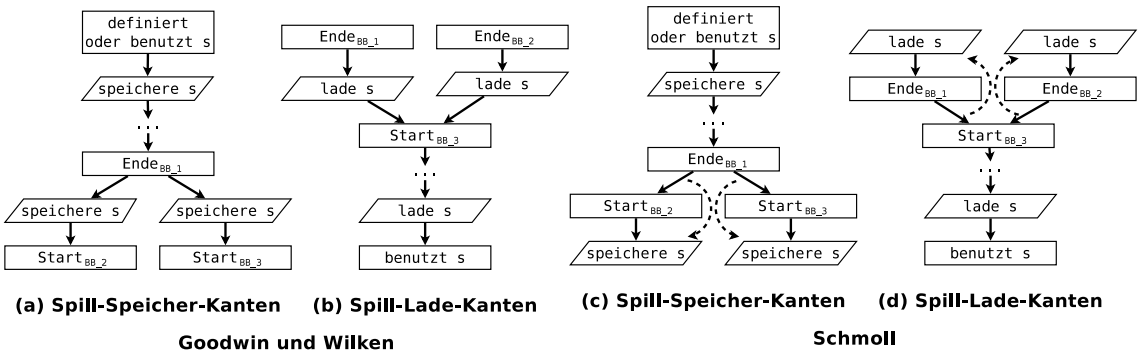


Abbildung 5.6: Unterschiedliche Positionierungen der Spillentscheidungen

Auf die Spill-Kanten der VRGs können jetzt die Transformationen angewendet werden, die in Abbildung 5.7 (a) und (b) dargestellt sind, um die Spillentscheidungen im ILP zu modellieren. Die Entscheidungsvariable $x^{st}_{s\to p}$ bzw. $x^{cont}_{s\to p}$ gibt an, ob eine Speicheroperationen eingefügt wird bzw. das virtuelle Register weiterhin zugeordnet bleibt. Beide können nur den Wert Eins erhalten, wenn das Register vorher allokiert war. Das wird durch die beiden dargestellten Nebenbedingungen sichergestellt. Die Entscheidungsvariable $x^{ld}_{s\to p}$ gibt an, ob eine Ladeinstruktion eingefügt wird, die das virtuelle Register s lädt. Dies ist wiederum nur möglich, wenn es zuvor durch eine Spillinstruktion in den Speicher geschrieben wurde. Deshalb muss auch der Zustand des Speichers durch einen **MG** (Memory Graph) modelliert werden.

Definition 5.3 (MG - Memory Graph) *Für jedes virtuelle Register s kann ein MG durch Anwendung der in Abbildung 5.7 (c) und (d) dargestellten Transformationsregeln aus dem zugehörigen IG erzeugt werden. Er enthält alle Knoten und Kanten, die auch der zugehörige IG enthält. Hinter den Kanten, an denen Spillentscheidungen getroffen werden müssen, werden ein neuer Knoten und eine neue Kante gemäß der Transformationsregeln eingefügt. Alle neu erzeugten Knoten werden mit Entscheidungsvariablen annotiert, die modellieren, ob die zugehörige Spillinstruktion eingefügt wird. Zusätzlich enthält der MG die Nebenbedingungen, die durch die Anwendung der Transformationsregeln entstehen und die den Zustand des Speichers modellieren, in den die virtuellen Register zwischengespeichert werden.*

Aus dem IG eines virtuellen Registers kann durch die Anwendung der Transformationsregeln, die in Abbildung 5.7 (c) und (d) dargestellt sind, sein MG erzeugt werden. Die Entscheidungsvariablen $x^{st}_{s\to p}$ und $x^{ld}_{s\to p}$ entsprechen denen, die bereits bei den Transformationen in Abbildung 5.7 (a) und (b) erzeugt wurden. Um sicher zu stellen, dass ein virtuelles Register vorher in den Speicher geschrieben wurde und es geladen werden kann, müssen die angegebenen Nebenbedingungen zusätzlich modelliert werden (Abbildung 5.7 d).

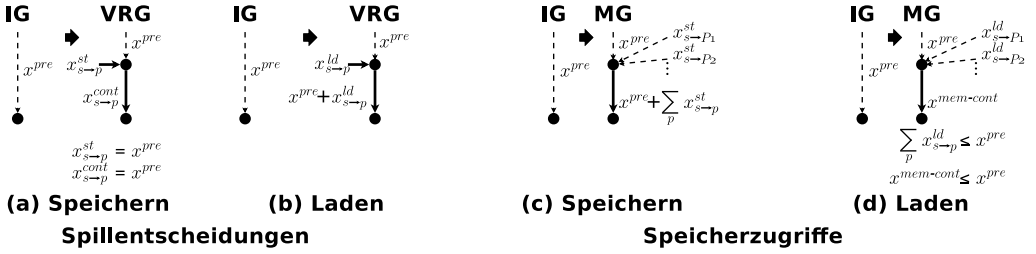


Abbildung 5.7: Speicher- und Ladetransformationen nach [GW96]

Die an den Kanten annotierten Summen (Abbildung 5.7 b und c) sollen andeuten, dass die Entscheidungsvariablen den Zustand der Register an der Kante wiedergeben. Folgt auf eine solche Kante ein Knoten, der tansforniert wird, und die Transformation erzeugt eine Nebenbedingung, die den Platzhalter x_{pre} verwendet, so wird der Platzhalter durch diese Summe ersetzt.

5.1.3 Modellierung interprozeduraler Eigenschaften

Die jeweils verwendeten Calling-Conventions können leicht in die bisherige Formulierung integriert werden. Allen Caller- und Callee-Saved-Registern werden zunächst virtuelle Register zugeordnet. Dies geschieht durch das Einfügen von Hilfsinstruktionen. Auf diese Weise kann für beide Arten von Registern der nötige Spillcode optimal eingefügt werden.

Den Caller-Saved-Registern darf zum Zeitpunkt eines Funktionsaufrufes kein virtuelles Register zugeordnet sein. Die entsprechenden Entscheidungsvariablen werden deshalb fest auf den Wert Null gesetzt.

Callee-Saved-Register müssen zu Beginn der Funktion gesichert und am Ende wieder geladen werden. Die entsprechenden virtuellen Register sind über die gesamte Programmlaufzeit lebendig und werden wie alle anderen virtuellen Register behandelt. Um die Zuordnung zu erhalten, werden die Register vorgefärbt. Das heißt, es werden nur die Entscheidungsvariablen für das entsprechende physikalische Register erstellt, so dass das virtuelle Register am Ende auf jeden Fall wieder dorthin geladen wird.

5.2 WCET-Modellierung

Wie zuvor erwähnt wurde, benötigt das modellierte ILP eine Zielfunktion. Für eine solche können verschiedene Optimierungsziele verwendet werden, wie z. B. die Codegröße, ACET oder der Energieverbrauch. Im Rahmen des WCC hingegen soll die WCET minimiert werden. Ziel soll es sein, den Spillcode-Overhead zu minimieren, so dass die WCET minimiert wird. Anders als bei den anderen Optimierungszielen wirken sich bei der WCET allerdings nur jene Spillinstruktionen auf die Zielfunktion aus, die auf dem WCEP liegen. Daher ist es notwendig, diesen entsprechend im ILP zu modellieren. Aufgrund der Instabilität des WCEP genügt es aber nicht, diesen alleine zu modellieren, daher müssen alle möglichen Ausführungspfade modelliert werden.

Im Abschnitt 2.4.1 wurde das IPET-Verfahren vorgestellt, welches die Modellierung aller möglichen Ausführungspfade und damit die Bestimmung der WCET ermöglicht. Diese Modellierung ist jedoch nicht als Zielfunktion für das im vorherigen Abschnitt vorgestellte ILP geeignet, weil es den WCEP bestimmt, indem es die maximale Ausführungsdauer über alle möglichen Pfade ermittelt. Es stellt somit ein Maximierungsproblem dar und modelliert die falsche Optimierungsrichtung. Deshalb benutzt Schmoll als geeignete Grundlage zur Modellierung der WCET in seiner Diplomarbeit jene, die Suhendra et al. in ihrem Verfahren [SMRC05] verwenden. Er erweitert den Ansatz, um auch komplexere Kontrollflussstrukturen modellieren zu können, bei denen das ursprüngliche Verfahren keine exakte Modellierung mehr zulässt.

5.2.1 WCET-Modellierung nach Suhendra et al.

Ausgehend vom CFG des zu modellierenden Programmes, der alle Basisblöcke enthält, werden durch Verschmelzungsschritte die Kosten aller Knoten zusammengefasst. Benötigt wird dazu das Wissen über die WCET_{est} aller Basisblöcke und die maximale Iterationszahl aller Schleifen. Die WCET_{est} eines Basisblocks B kann durch die ganzzahlige Variable d_B ausgedrückt werden, die anhand der folgenden Nebenbedingung im ILP modelliert werden kann:

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

Die ganzzahlige Variable g_B gibt die konstanten Grundkosten des Basisblockes B ohne eingefügten Spillcode wieder. Die Ermittlung dieser Grundkosten wird in Abschnitt 5.2.4 beschrieben. Die nachfolgende Summe repräsentiert den Spillcode-Overhead des Basisblocks. Die ganzzahligen Kostenvariablen $c(x_B^k)$ der jeweiligen Spillentscheidungen werden mit den Entscheidungsvariablen x_B^k multipliziert. Diese geben an, ob die zugehörige Spillinstruktion eingefügt wird, x_B^k steht dabei als Platzhalter für eine der zuvor erwähnten Variablen $x_{s\to p}^{st}$ oder $x_{s\to p}^{ld}$.

Zunächst werden beginnend bei den inneren die Ausführungsdauern aller Schleifen modelliert. Jede Schleife z erhält eine ganzzahlige Variable g_z, die die maximale Ausführungsdauer angibt. Die Schleife wird dann im CFG durch einen Ersatzknoten ausgetauscht, der diese Variable als Kosten zugewiesen bekommt. Auf diese Weise können alle Verschachtelungen von Schleifen von innen heraus schrittweise ersetzt werden. Existieren mehrere Schleifenausgänge, wird ein Hilfsknoten mit dem WCET_{est}-Wert Null eingefügt. Zusätzlich werden auf ihn gerichtete Kanten eingefügt, die von den bisherigen Ausgängen ausgehen.

Mit Hilfe der nachfolgenden Reduktionsschritte kann die Schleife von der Senke zur Quelle hin induktiv modelliert werden. Jeder Knoten des CFG, der eine Instruktion innerhalb der Schleife darstellt, erhält eine ganzzahlige Variable w_B, die die WCET modelliert, die er und seine Nachfolger verursachen. Die Senke erhält dabei direkt die WCET_{est} des zugehörigen Basisblockes, was durch die folgende Nebenbedingung ausgedrückt wird.

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

Für alle vorhergehenden Knoten B gilt, dass für jeden Nachfolger N eine Nebenbedingung eingefügt werden muss, um alle verschiedenen Ausführungspfade zu berücksichtigen.

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

Nach der Modellierung der WCET einer Schleife z kann diese durch einen Ersatzknoten Z mit Kosten g_z ausgetauscht werden. Diese entspricht dem Produkt aus WCET des Schleifenanfangsknotens und der maximalen Iterationszahl lb_z. Ersterer entspricht der Quelle des Teilgraphen, der die Schleife z darstellt.

$$g_z = lb_z \cdot w_{Quelle} \tag{5.4}$$

Sind alle Schleifen auf diese Weise ersetzt worden, ist der resultierende CFG anschließend zyklenfrei. Der gesamte Kontrollfluss kann nun mit Hilfe der Nebenbedingungen 5.2 und 5.3 induktiv modelliert werden, wie dies auch bei Schleifen geschieht. Beim Vorhandensein mehrerer Senken wird ebenfalls analog verfahren.

Diese Modellierung erzeugt eine untere Schranke für die $\mathrm{WCET}_{\mathrm{est}}$, die definitionsgemäß über der tatsächlichen WCET liegt. Als Zielfunktion wird nun die ganzzahlige Variable w_I benutzt, die die WCET des initialen Knotens des CFG modelliert. Durch Minimierung dieser Zielfunktion wird erzwungen, dass die modellierte WCET den Wert der unteren Schranke annimmt.

Laut Schmoll besitzt diese Modellierung aber eine Schwäche. Sie ersetzt demnach Schleifen zu rigoros und modelliert dabei Pfade mit, die eigentlich nicht existieren. Eine korrekte Modellierung entsteht nur, wenn alle Schleifen einen Eingang und Ausgang besitzen. Er erweitert diese Modellierung deshalb um die Darstellung reduzierbarer Graphen, wie dies im folgenden Abschnitt erläutert wird.

5.2.2 WCET-Modellierung für reduzierbare Graphen

Schmoll adaptiert das Verfahren aus Abschnitt 5.2.1, um es für reduzierbare Graphen anzupassen. In reduzierbaren Graphen weist jede Schleife genau einen Eingang auf. Sie kann jedoch über mehrere Ausgänge verfügen. Des Weiteren gilt, dass verschachtelte Schleifen nur in der richtigen Reihenfolge hintereinander betreten werden können. Beim Verlassen können hingegen mehrere Schleifen auf einmal verlassen werden. Enthält eine Programmiersprache nur Sprachelemente wie if-then-else, while-do, continue und break, so besitzen alle Programme auf jeden Fall einen reduzierbaren CFG. Alleine das Verbieten von goto-Anweisungen genügt nicht als Voraussetzung für das Vorhandensein von reduzierbaren CFGs, wie Duff's Device zeigt [Duf88]. Irreduzierbare Graphen können jedoch durch ein Node-Splitting in reduzierbare Graphen überführt werden.

Wie schon in Abschnitt 5.2.1 angeführt wurde, muss für alle Basisblöcke die $\mathrm{WCET}_{\mathrm{est}}$ und für alle Schleifen die maximale Iterationszahl lb vorliegen. Die Modellierung der Variable d_B für jeden Basisblock B bleibt unverändert. Eine Änderung tritt bei der Modellierung der Schleifen auf, wo jetzt mehrere Ausgänge existieren können. Den ersten $lb - 1$ Iterationen einer Schleife wird die WCET des Pfades mit der größten WCET innerhalb der Schleife zugeordnet. Die letzte Iteration erhält die WCET des vom Schleifenanfang bis zum jeweils benutzten Ausgang führenden

Pfades. Um die verschiedenen Ausgänge der Schleife L darstellen zu können, aber auch aufgrund der Tatsache, dass nicht entschieden werden kann, welcher Ausgang auf dem WCEP liegt, muss für jeden Ausgang A eine separate ganzzahlige Variable g_A^L erzeugt werden.

Um verschiedene Pfade für die ersten $lb-1$ Iteration und die letzte Iteration einer Schleife modellieren zu können, verwendet Schmoll das Verfahren des **Schleifenschälens (Loop Peeling)** aus [BGS94].

Dabei werden Ausgänge gesucht, nach denen nicht wieder zum Schleifenanfang gesprungen wird. Alle Knoten, die auf dem Pfad vom Schleifenanfang zu einem dieser Ausgänge liegen, werden kopiert, also insbesondere auch die Knoten, die den Schleifenanfang und -ausgang repräsentieren. Der neu entstehende Teilgraph wird als Schale der Schleife bezeichnet (Abbildung 5.8 a und b). Die kopierten Knoten repräsentieren dieselben Basisblöcke wie ihre Originale und werden auf dieselbe Weise untereinander verbunden. Die ausgehenden Kanten der zuvor gefundenen Ausgänge werden entfernt. Alle Ausgänge, bei denen wieder zum Schleifenanfang gesprungen wird, erhalten eine Kante auf die Kopie des Schleifenanfangsknotens. Die Iterationshäufigkeit der ursprünglichen Schleife wird um Eins verringert.

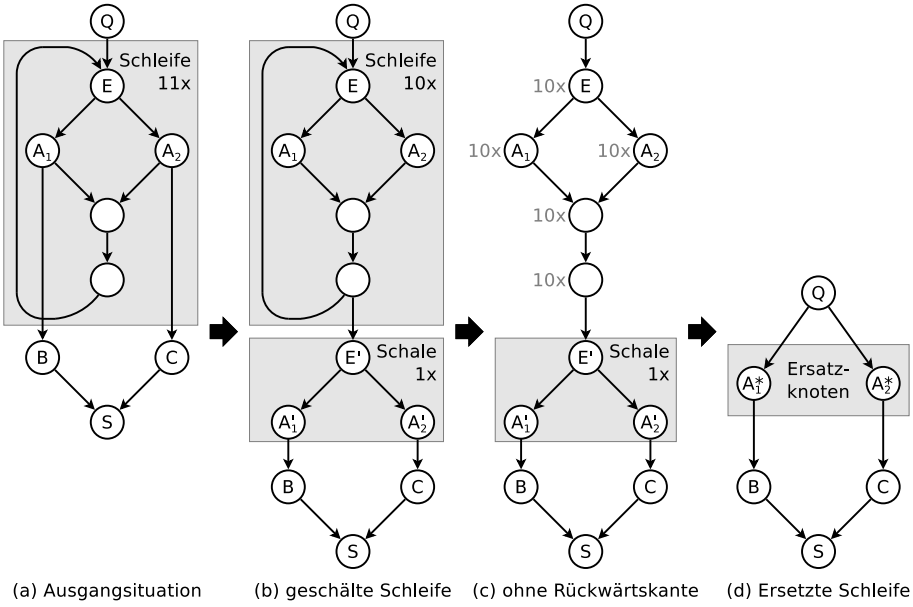


Abbildung 5.8: Schleifenschälen nach [Sch08]

Die Schleife und ihre Schale werden aber weiterhin als Einheit betrachtet und können zusammen durch Ersatzknoten ausgetauscht werden. Der Unterschied liegt darin, dass nun für jeden Ausgang A ein Ersatzknoten A_{Ersatz} erzeugt wird (Abbildung 5.8 d). Um die Modellierungstechnik aus dem vorherigen Abschnitt verwenden zu können, muss die Rückwärtskante der Schleife entfernt werden (Abbildung 5.8 c). Diese kann einfach weggelassen werden, wenn jeder Basisblock B innerhalb der Schleife die maximale Iterationszahl der Schleife weniger Eins als Ausführungshäu-

figkeit $It(B)$ erhält. Diese Ausführungshäufigkeiten der Schleifen werden im WCC über eine Schleifenanalyse ermittelt oder durch Flow-Facts angegeben. Auf diese Weise werden die Ersetzungsschritte wie zuvor durchgeführt, wobei wiederum die Schleifen von innen nach außen ersetzt werden. Jeder Ersatzknoten A_{Ersatz} stellt im Endeffekt die WCET der Schleife dar, die aus $lb-1$ Schleifendurchläufen und einem Pfad innerhalb der Schleifenschale hin zum jeweiligen Ausgang besteht.

Anders als bei Suhendra et al. geht Schmoll so vor, dass er die WCET von der Quelle ausgehend induktiv modelliert. Durch die Modellierung mehrerer Ausgänge ist eine Umkehrung der Reihenfolge sinnvoll, da so bereits gewonnene Informationen leichter weiterverwendet werden können. Deswegen werden auch die Schleifen induktiv vom Schleifenanfangsknoten aus modelliert. Die WCET dieses Knotens wird durch die ganzzahlige Variable m_{Quelle} modelliert.

$$m_{Quelle} = It(Quelle) \cdot d_{Quelle} \tag{5.5}$$

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

Die nachfolgenden Knoten innerhalb der Schleife werden darauf aufbauend induktiv modelliert, bis eine Senke erreicht wird. Für Basisblöcke C mit einem Vorgänger B bzw. mehreren Vorgängern B wird die Nebenbedingung 5.7 bzw. 5.8 eingefügt.

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

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

Schlussendlich entspricht die Variable, die den jeweiligen Ausgang der Schleife L modelliert, der WCET der Kopie A' des zugehörigen Basisblockes A.

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

Nachdem die WCET aller Schleifen modelliert wurde und diese durch Ersatzknoten ausgetauscht wurden, liegt ein azyklischer CFG vor. Mit Hilfe der Nebenbedingungen 5.7 und 5.8 kann nun die WCET der gesamten Funktion induktiv modelliert werden. Die Funktion $It(C)$ liefert jetzt für alle Basisblöcke den Wert Eins, da kein vorhandener Basisblock mehrfach ausgeführt werden kann.

Für alle Senken D des CFG, die jeweils ein Funktionsende der Funktion F darstellen, muss zum Schluss noch eine Nebenbedingung eingefügt werden.

$$\forall Senken\, D : m_F \geq m_D \tag{5.10}$$

Die Variable m_F modelliert die WCET der gesamten Funktion und stellt somit die Zielfunktion dar. Wird m_F minimiert, so nimmt es den Wert der kleinsten unteren Schranke der tatsächlichen WCET an. Es entspricht einem Maximum über allen modellierten WCET-Werten, die durch die Variablen m_D repräsentiert werden.

5.2.3 Modellierung interprozeduraler Eigenschaften

Da auch Funktionsaufrufe in die Modellierung der WCET mit einfließen sollen, muss die Aufrufreihenfolge bekannt sein. Funktionen, die keine anderen Funktionen aufrufen, werden zuerst modelliert. Ihre WCET_{est} kann in der Modellierung der aufrufenden Funktionen somit einfließen.

Eine solche Modellierung eine Einschränkung notwendig. So sind Rekursionen nicht zugelassen, da die jeweils aufgerufene Funktion noch nicht modelliert werden konnte. Alternativ kann der Funktionsaufruf ignoriert werden und Rekursionen können doch zugelassen werden.

5.2.4 Ermittlung der WCET-Daten

Zur Modellierung der WCET eines Basisblocks werden dessen Grundkosten g_B benötigt. Diese konstanten Grundkosten bestehen aus der WCET_{est} des Basisblocks, die im WCC über das Analyseprogramm a^3 mittels einer statischen WCET-Analyse ermittelt wird. Diese WCET_{est} gibt die maximale Ausführungsdauer des Basisblocks ohne eingefügten Spillcode an. Daraus ergibt sich ein Problem, da ausführbarer Programmcode vorliegen muss, der analysiert werden kann. Damit das Programm allerdings ausführbar ist, muss eine vorläufige Registerallokation durchgeführt werden, die **Vorallokation** genannt wird. Dazu kann ein beliebiges anderes Registerallokationsverfahren benutzt werden. Schmoll verwendet an dieser Stelle den im WCC vorhandenen naiven Allokator RASA (Abschnitt 3.2.2). Dieser erzeugt eine Zuordnung der Register, aber er fügt auch Spillcode ein. Um die Grundkosten bestimmen zu können, müssen die Kosten für die Spillinstruktionen wieder aus den ermittelten WCET_{est}-Werten herausgerechnet werden.

Schmoll versucht in seiner Diplomarbeit, den zusätzlich erzeugten Spillcode-Overhead jedes Basisblocks in der Vorallokation abzuschätzen. Er trifft dazu vier Annahmen, die Einschränkungen darstellen und somit eine Erleichterung bei der Implementierung darstellen:

1. Lade- und Speicherbefehle weisen bzgl. der WCET dasselbe Verhalten auf.
2. Die Kosten einer Spillinstruktion hängen nur von deren Typ und niemals vom Kontext ab, in den diese eingefügt wird.
3. Ein konstanter Kostenwert von Eins kommt den tatsächlichen Kosten einer Spillinstruktion sehr nahe, da sich die möglichen Vorteile einer möglichen Parallelausführung und die Wartezeiten bei Cache-Misses in etwa neutralisieren.

4. Alle Register des TC1796 sind gleichwertig. Die Ausführungszeit eines Basisblocks hängt demnach nie von der Registerzuordnung ab. Unterschiede in der Ausführungszeit eines Basisblocks nach einer Vorallokation werden nur durch Spillinstruktionen hervorgerufen.

Das erste Schätzverfahren betrachtet die Anzahl der Spillinstruktionen s_B und ursprünglichen Instruktionen n_B im aktuellen Basisblock und versucht daraus den Anteil der Kosten abzuwägen, den die Spillinstruktion verursacht haben. Die von a^3 ermittelte $\mathrm{WCET}_{\mathrm{est}}$ wird dazu mit einem Faktor gewichtet.

$$g_B = \mathrm{WCET}^B_{\mathrm{est}} \cdot \frac{n_B}{n_B + s_B} \tag{5.11}$$

Diese Schätzung nennt Schmoll auch **einfache Schätzung**. In seiner **aufwändigen Kostenschätzung** verwendet er die $\mathrm{BCET}_{\mathrm{est}}$, die er separat schätzt. Die durch Spillinstruktionen auftretenden Verzögerungen versucht er genauer zu erfassen. Er schätzt dazu einmal die $\mathrm{BCET}_{\mathrm{est}}$ des Basisblocks mit ($\mathrm{BCET}^B_{\mathrm{spill}}$) und ohne ($\mathrm{BCET}^B_{\mathrm{ohne}}$) Spillinstruktionen. Eine schlechte $\mathrm{BCET}_{\mathrm{est}}$ sorgt in diesem Fall dafür, dass das Ergebnis sich dem der einfachen Schätzung annähert oder sogar mit diesem übereinstimmt. Diese Art der Schätzung soll bessere Ergebnisse liefern, wenn die Kosten von Spillinstruktionen z. B. durch Parallelausführungen vom Wert Eins abweichen.

$$g_B = (\mathrm{WCET}^B_{\mathrm{est}} - \mathrm{BCET}^B_{\mathrm{spill}}) \cdot \frac{n_B}{n_B + s_B} + \mathrm{BCET}^B_{\mathrm{ohne}} \tag{5.12}$$

Eine präzise Ermittlung der $\mathrm{WCET}_{\mathrm{est}}$ des Basisblocks ohne Spillcode kann mit Hilfe dieser Schätzungen nicht durchgeführt werden. Es gibt des Weiteren keine Garantie, dass die Schätzung eine sichere obere Schranke für die WCET darstellt. Da die ermittelten Schätzwerte allerdings nur zur Modellierung benutzt werden, ist dies auch nicht zwingend erforderlich. Nach der eigentlichen Registerallokation muss deshalb eine erneute WCET-Analyse durchgeführt werden, um die Güte der Registerallokation verlässlich bestimmen zu können.

5.3 Zusammenfassung der bisherigen Implementierung

In Abbildung 5.9 ist der gesamte Ablauf der Registerallokation noch einmal dargestellt. Diese Abbildung stellt einen Ausschnitt aus der Abbildung 3.2 dar, der das Verfahren der Registerallokation detailiert veranschaulicht. Als Eingabe erhält die Registerallokation den Programmcode als ICD-LLIR, die vom LLIR-Codeselektor erzeugt wurde. Das Verfahren selbst beginnt mit der Erstellung einer Vorallokation auf der ICD-LLIR, damit der Programmcode ausführbar wird und vom WCET-Analyseprogramm a^3 ausgewertet werden kann. Die $\mathrm{WCET}_{\mathrm{est}}$ -Werte der einzelnen

Basisblöcke liegen anschließend vor. Sie enthalten aber zusätzlich die Kosten des Spillcodes, welche noch subtrahiert werden müssen. Diese werden jedoch nicht bestimmt und subtrahiert, sondern sie werden geschätzt und über einen Faktor herausgerechnet (Abschnitt 5.2.4), weshalb es sich hierbei um ein heuristisches Verfahren handelt. Die geschätzten Kosten der Basisblöcke ohne Spillcode können nun benutzt werden, um das ILP gemäß der Abschnitte 5.1 und 5.2 zu modellieren. Das entstandene ILP wird daraufhin mit Hilfe des ILP-Solvers CPLEX gelöst. Die Registerzuordnungen und Einfügungen von Spillcode werden dann auf Grundlage der Werte der Entscheidungsvariablen durchgeführt und in die ICD-LLIR übertragen. Die fertig allokierte ICD-LLIR mit Registerzuordnungen und eingefügtem Spillcode wird ansschließend weitergegeben und steht anderen Optimierungen oder dem Codegenerator zu Verfügung.

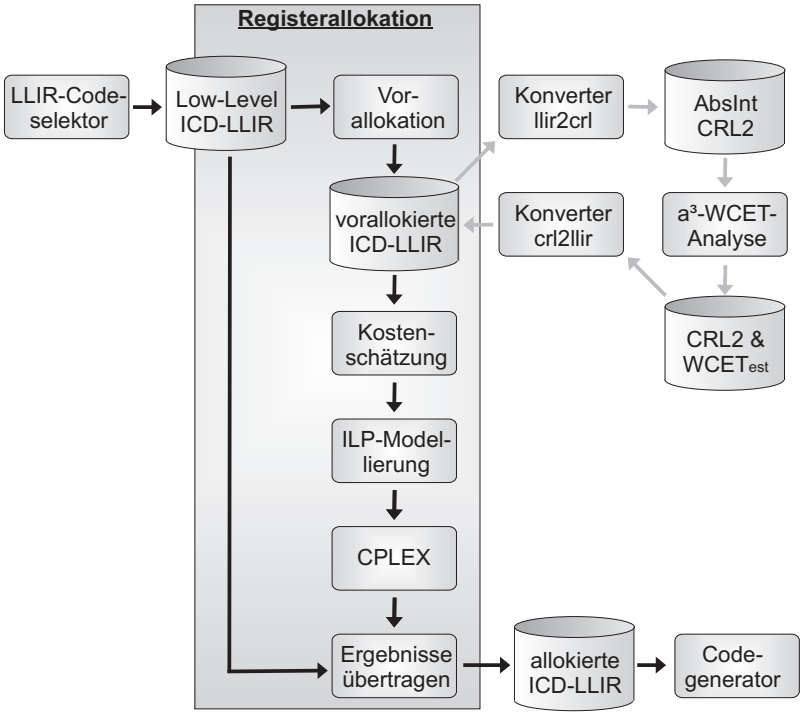


Abbildung 5.9: Schema der Registerallokation von Schmoll

5.4 Schwachpunkte der bisherigen Implementierung

Der von Schmoll entwickelte Registerallokator reduziert die WCET im Vergleich zum graphfärbungsbasierten Registerallokator RAGC i. d. R. deutlich, aber einige wenige Benchmarks erzeugen hingegen auch starke Verschlechterungen. Der codegrößenoptimierende Allokator RAILP-CS liefert für fast alle verwendeten Benchmarks bessere Ergebnisse bzgl. der WCET. Auch eine Variation der Kostenschätzung und der konstanten Kosten für die Spillinstruktionen bringt keine nennenswerten Vorteile. Die

Ergebnisse sind für einen WCET-optimierenden Registerallokator betrachtet jedenfalls nicht zufriedenstellend. Vor allem, da ein anderes Verfahren deutlich besser abschneidet, das keinerlei Annahmen bzgl. der WCET macht.

In der Zusammenfassung seiner Diplomarbeit nennt Schmoll einige Schwachstellen, die er in seiner Implementierung ausgemacht hat, und gibt eine ganze Reihe von Verbesserungsansätzen vor.

1. Er führt an, dass die WCET_{est}-Schätzung für Basisblöcke zu ungenau ist und der WCEP aufgrund dessen nicht genau genug modelliert werden kann. Diese Einschätzung bezieht sich auf beide zuvor in Abschnitt 5.2.4 erwähnten Schätzverfahren.
2. Die ersten beiden Einschränkungen der Aufzählung in Abschnitt 5.2.4 sind problematisch, da Lade- und Speicherbefehle an verschiedenen Stellen im Programm eingefügt werden. Zudem verhalten sie sich auch unterschiedlich, wenn sie auf dem TC1796 ausgeführt werden. Ein eingefügter Speicherbefehl kann i. d. R. öfter parallel zu einem anderen Befehl ausgeführt werden.
3. Auch die dritte Einschränkung, dass für die Modellierung ein konstanter Kostenwert für alle Spillinstruktionen genutzt wird, identifiziert er als Schwachstelle seiner Implementierung. Er schlägt vor, den Kostenwert variabel zu gestalten und das Wissen auch in die Schätzung der WCET_{est} zu übernehmen.
4. Als weitere Möglichkeit nennt er die Kombination mit einem weiteren Optimierungsziel wie z. B. der Codegröße, wobei die WCET-basierte Modellierung stärker gewichtet werden soll, um das primäre Optimierungsziel zu bleiben.
5. Auch die Einbeziehung weiterer Flow-Facts führt er an. So könnten etwa nicht ausführbare Pfade aus der Modellierung entfernt werden, damit nicht zufällig ein WCEP minimiert wird, der eigentlich gar nicht ausführbar ist.
6. Die hohe Rechenzeit, die das Verfahren teilweise benötigt, kann durch den Ansatz von Fu und Wilken [FW02] reduziert werden. Dieser entfernt Entscheidungsvariablen, die keine Auswirkung auf die Optimalität des Ergebnisses haben. Eine Steigerung der Güte kann so hingegen nicht erreicht werden.
7. Als letzte Möglichkeit wird vorgeschlagen, das für die Vorallokation benötigte Registerallokationsverfahren auszutauschen, da der verwendete naive Registerallokator eine genauere Schätzung der WCET_{est} vielleicht behindert. Er schlägt hier aufgrund der geringen Laufzeit und der Tatsache, dass weitaus weniger Spillcode eingefügt wird, ein Linear-Scan-Verfahren vor.

5.5 Lösungsansätze

Dieser Abschnitt stellt den Übergang von den Arbeiten Schmolls zu den im Rahmen dieser Diplomarbeit vorgenommenen Implementierungen und Erweiterungen

dar. Das Hauptziel besteht in der Steigerung der Güte des schon vorhandenen Registerallokators. Dazu werden einige der Punkte aufgegriffen, die in Abschnitt 5.4 aufgezählt werden. Deswegen wird nachfolgend in der selben Reihenfolge aufgeführt, ob der vorgeschlagene Ansatz im Rahmen dieser Diplomarbeit behandelt und in welchem Abschnitt das Vorgehen beschrieben wird.

1. Im Abschnitt 6.4 wird ein statisches Analyseverfahren beschrieben, das die Ermittlung der Mehrkosten ermöglicht, die durch Spillinstruktionen in einem Basisblock erzeugt werden. Dieses Verfahren benutzt Wissen über komplexe Zusammenhänge des Pipelineverhaltens der TC1796-Architektur.
2. Das unterschiedliche Verhalten von Lade- und Speicherinstruktionen beim TC1796, insbesondere im Zusammenhang mit dem Ausführungskontext, wird in beiden im Kapitel 6 behandelten pipelinebasierten Verfahren explizit berücksichtigt.
3. Sowohl in der pipelinebasierten Kostenschätzung in Abschnitt 6.4 wird die Verwendung variabler Kosten für jede Spillinstruktion umgesetzt, als auch in der Erweiterung der ILP-Modellierung aus Abschnitt 6.5.
4. Eine Kombination mit einem codegrößenoptimierenden Ansatz wird in Abschnitt 7.4 skizziert.
5. Eine Einbeziehung von Informationen über nicht ausführbare Pfade in die Modellierung wird nicht vorgenommen.
6. Die Umsetzung des Ansatzes von Fu und Wilken [FW02] zur Beschleunigung des Verfahrens wird in dieser Diplomarbeit nicht berücksichtigt, da er keine Steigerung der Güte ermöglicht, sondern ausschließlich eine Reduzierung der benötigten Rechenzeit.
7. Auch verschiedene Vorallokationen werden im Rahmen dieser Diplomarbeit auf ihre Auswirkung hin in Abschnitt 7.2 getestet.

Zusätzlich wird als weitere Maßnahme der Programmcode während der WCET-Analyse dem SPM zugeordnet, die Gründe und Auswirkungen werden in Abschnitt 7.1 näher beschrieben. Darüber hinaus wird die Optimierung der Copy-Elimination im Abschnitt 7.3 in das ILP integriert, ähnlich wie sie Goodwin und Wilken in [GW96] modellieren.

„Der superskalare Prozessorkern besteht aus zwei Hauptpipelines mit jeweils vier Stufen und einer Nebenpipeline zur Schleifensteuerung. Die drei Pipelines arbeiten parallel und erlauben es drei Instruktionen in einem Zyklus auszuführen.“

Infineon Technologies AG
übersetzt aus dem Englischen [Inf02, Seite 40]

6 Pipelineanalyse

Dieses Kapitel greift die Punkte 1 bis 3 aus Abschnitt 5.4 auf, die Schmoll selber als Schwachpunkte seines Verfahrens ausgemacht hat. Zum einen führt er an, dass beide Kostenschätzverfahren zu ungenau sind, um den WCEP präzise modellieren zu können, und zum anderen trifft er eine Reihe von Entscheidungen, die seine Implementierung vereinfachen, aber eine gewisse Ungenauigkeit verursachen. So geht er davon aus, dass sich Lade- und Speicherinstruktionen gleich verhalten und der Ausführungskontext keine Auswirkung auf die Kosten einer eingefügten Spillinstruktion haben soll. Zusätzlich wählt er einen konstanten Kostenwert, der für alle Spillinstruktionen den gleichen Wert hat.

Betrachtet man das Pipelineverhalten des TC1796 in Abschnitt 4.5 und die Eigenschaften des zugehörigen Befehlssatzes (Abschnitt 4.4), insbesondere die Instruktionseigenschaften in den Tabellen 4.1 und 4.3, so wird deutlich, dass die zuvor genannten Vereinfachungen zu einer Modellierung führen, die dem Verhalten des TC1796 nicht gerecht werden. Spillinstruktionen werden an den Spill-Lade-Kanten und Spill-Speicher-Kanten eingefügt (Abbildung 5.6 c und d). Die eingefügten Lade- und Speicherbefehle können sich unterschiedlich verhalten und auch verschiedene Kosten verursachen, was mit den Stellen zusammenhängt, an denen sie eingefügt werden, weil die Kosten in direktem Zusammenhang mit dem Ausführungskontext stehen.

Diese Diplomarbeit verfolgt deshalb das Ziel, explizites Wissen über das Pipelineverhalten in die bisherige Implementierung der Registerallokation zu integrieren. Einige Beispiele, die zu den Erweiterungen in diesem Kapitel motivieren, werden im folgenden Abschnitt aufgezeigt. Im Weiteren werden zwei Verfahren dargestellt, deren Grundprinzip sich ähnelt und daher im Vorfeld in Abschnitt 6.2 vorgestellt wird. Die Integration der Verfahren in den bisherigen Ansatz wird in Abschnitt 6.3 erläutert. In dem ersten der beiden Verfahren sollen die Basisblockkosten aus der WCET-Analyse der Vorallokation genauer von den Kosten der eingefügten Spillinstruktionen bereinigt werden, damit die Modellierung des WCEP präziser vonstattengehen kann. Dieses Verfahren der statischen Pipelineanalyse wird in Abschnitt 6.4 im Detail beschrieben. Das zweite Verfahren aus Abschnitt 6.5 stellt im eigentlichen Sinne eine Erweiterung des bisherigen ILPs dar, welches um weitere Nebenbedingungen erweitert wird. Die Elemente der Pipelineanalyse werden genutzt, um die Kosten einer Spillinstruktion individuell modellieren zu können. Es heißt daher im

Folgenden dynamische Pipelineanalyse, da die Auswertung des Pipelineverhaltens während der Lösung des ILPs geschieht. An dieser Stelle sei noch darauf hingewiesen, dass immer von Schmolls Registerallokator bzw. dessen Erweiterung die Rede ist, wenn der Begriff Registerallokator verwendet wird. Alle anderen erwähnten Registerallokatoren werden deutlich erkennbar benannt.

6.1 Motivation

Innerhalb des Registerallokators werden ausschließlich Spillinstruktionen in Form von 32-Bit-Lade- und Speicherbefehlen eingefügt. Diese können parallel zu vorherstehenden IP-Instruktionen ausgeführt werden, wie die Abbildung 4.4 und die Tabelle 4.3 zeigen. Ein Speicherbefehl kann immer parallel zu einem vorhergehenden IP-Befehl und damit kostenlos ausgeführt werden. Ladebefehle besitzen die Einschränkung, dass sie datenunabhängig zum IP-Befehl davor sein müssen, um parallel ausgeführt werden zu können. Zudem sind Speicherbefehle im Gegensatz zu Ladebefehlen nicht blockierend. In Abhängigkeit von dem Speicher, in dem die virtuellen Register abgelegt werden, und dessen Latenz können Ladebefehle somit deutlich teurer als Speicherbefehle sein. Die virtuellen Register werden durch die Registerallokation auf dem Laufzeitstack abgelegt. Dieser wiederum befindet sich im SRAM innerhalb der DMU (Abbildung 4.1) und besitzt eine Latenz von Eins. Die zu ladenden Registerinhalte stehen somit bereits im nächsten Zyklus zur Verfügung, weshalb Ladebefehle entweder Kosten Null oder Eins verursachen, je nachdem, ob sie parallel zum Vorgängerbefehl ausgeführt werden können oder nicht.

Eine Eigenschaft, die die Pipelineanalyse noch zusätzlich verkompliziert ist, dass sogar zwei bzw. drei datenunabhängige Lade- und Speicherbefehle parallel zu einem IP-Befehl ausgeführt werden können, wenn dessen Latenz den Wert Zwei bzw. Drei hat (Tabelle 4.1). Zudem kann auch Parallelität durch das Einfügen von Spillinstruktionen verloren gehen. Zwei Mehrzyklen-IP-Befehle mit demselben Befehlscode, die zuvor direkt hintereinander ausgeführt wurden, können nach der Registerallokation durch eine datenabhängige Spillinstruktion getrennt sein. Der zweite IP-Befehl kann infolgedessen nicht mehr die Wiederholungsrate ausnutzen, er benötigt daher als Ausführungszeit wieder die Latenz.

1. WAW-Konflikt zwischen einem IP- und Ladebefehl
2. RAW-Konflikt zwischen einem Lade- und LS-Befehl
3. Ein Speicher- und Ladebefehl greifen auf die selbe Adresse zu
4. Ein LS-Befehl folgt auf einen Funktionsaufruf oder Rücksprungbefehl
5. Ein Speicherbefehl steht vor einem Rücksprungbefehl
6. Konflikt zwischen einem Mehrzyklen-IP- und einem Speicher- oder Ladebefehl

Tabelle 6.1: Zusammenfassung der Stall-Gründe aus Abschnitt 4.5

Im Abschnitt 4.5 wurden sechs Situationen aufgezählt, die Konflikte verursachen und zu Stalls innerhalb der Pipeline führen. Der erste und sechste Fall betreffen Speicher- und Ladebefehle, die, wie schon oben erwähnt, parallel zu einem vorhergehenden IP-Befehl ausgeführt werden, wenn keine Datenabhängigkeit, das heißt kein RAW bzw. WAW, vorliegt.

Der zweite Fall hingegen bezieht sich nur auf Ladebefehle. Hinter diesen wird ein Stallzyklus eingefügt, wenn ein nachfolgender LS-Befehl ein geladenes Adressregister benutzt. Beim dritten Fall kommt es zu einer Wechselwirkung zwischen einem Speicherbefehl und einem nachfolgenden Ladebefehl, wenn beide auf die selbe Adresse in einem der darüberliegenden Speicher zugreifen und in Folge dessen ein Stallzyklus entsteht. Die vierte Situation betrifft alle Arten von LS-Befehlen, die direkt auf einen Funktionsaufruf oder einen Rücksprungbefehl folgen, da diese die LS-Pipeline noch einen Zyklus belegen. Ähnlich verhält es sich beim fünften Fall, bei dem ein direkt vor einem Rücksprung stehender Speicherbefehl erst komplett abgearbeitet werden muss, bevor der Rücksprung ausgeführt werden kann, was ebenfalls zu einem Stallzyklus führt.

Die zuvor genannten Eigenschaften des Pipelineverhaltens des TC1796 beeinflussen die Kosten von Spillinstruktionen. Diese können unter Umständen kostenlos ausgeführt werden, sie können aber auch zusätzliche Kosten verursachen, wenn sie nicht parallel ausgeführt werden können. Zudem können sie weitere Stalls verursachen und so noch höhere Kosten verursachen. Es sind aber sogar Fälle denkbar, in denen Spillinstruktionen nicht parallel zum Vorgängerbefehl ausgeführt werden können, sie aber trotzdem kostenlos eingefügt werden können, weil sie einen ursprünglich vorhandenen Konflikt zwischen originalen Befehlen beheben. Generell können Spillinstruktion, die von der Registerallokation unter den oben genannten Eigenschaften eingefügt werden, Kosten von null bis drei zusätzlichen Zyklen verursachen.

6.2 Prinzip der Pipelineanalyse

Das Prinzip der hier behandelten Pipelineanalyseverfahren liegt in der Einteilung der durch das Einfügen von Spillinstruktionen entstehenden Kosten oder Einsparungen in vier Kategorien. Diese erhalten jeweils eine Variable, die die Anzahl der zusätzlichen benötigten oder eingesparten Zyklen wiedergibt.

Direkte Kosten: Diese Kosten hängen alleine von den Vorgängern der Spillinstruktion ab und geben an, ob die Spillinstruktion parallel ausgeführt werden kann. Berücksichtigt werden hierfür der erste und sechste Fall aus Tabelle 6.1. Die direkten Kosten c_d betragen Null, wenn eine Parallelausführung möglich ist, anderenfalls betragen sie Eins und geben somit an, dass die entsprechende Spillinstruktion einen separaten Zyklus zur Abarbeitung benötigt. Zwei Eigenschaften des TC1796 machen die Bestimmung von c_d schwierig. Zum einen das Vorhandensein von Mehrzyklenbefehlen und zum anderen die mögliche Ausnutzung der Wiederholungsrate.

Es genügt also nicht, nur den direkten Vorgänger der Spillinstruktion zu betrachten. Man muss herausfinden, wo der nächste IP-Befehl vor der zu betrachtenden Spillinstruktion steht. Anschließend muss überprüft werden, ob die Ausführungszeit des IP-Befehles hoch genug ist, dass die Spillinstruktion parallel ausgeführt werden könnte. Insgesamt genügt es dazu, die drei vorhergehenden Befehle zu betrachten, da kein IP-Befehl eine Latenz größer Drei besitzt. Könnte die Spillinstruktion theoretisch parallel zum gefundenen IP-Befehl ausgeführt werden, so muss man sich nun vergewissern, dass zwischen den beiden Befehlen ausschließlich LS-Befehle liegen, die zum IP-Befehl datenunabhängig sind. In diesem Fall wäre auch die Ausführung der Spillinstruktion kostenlos.

Die Ausführungszeit, die ein IP-Befehl innerhalb der Execute-Stufe verbringt, entspricht entweder der Latenz oder der Wiederholungsrate aus Tabelle 4.1. Zusätzlich muss auch noch der nächste IP-Befehl vor dem eigentlich betrachteten IP-Befehl untersucht werden. Hat dieser Vorgänger keinen Konflikt zum IP-Befehl, und zwischen beiden liegen nur LS-Befehle, die parallel zum Vorgänger ausgeführt werden können, so benötigt der IP-Befehl nur so viele Zyklen in der Execute-Stufe, wie die Wiederholungsrate angibt. Dieser Sachverhalt verkompliziert der Ermittlung der direkten Kosten erheblich.

Stall-Kosten: Die Stall-Kosten c_s orientieren sich an den in Tabelle 6.1 erwähnten Fällen 2 bis 5. Aufgrund möglicher Parallelausführungen müssen je nach Fall dazu ein oder zwei Vorgänger- oder Nachfolgebefehle betrachtet werden. Für jeden Konflikt, der auftritt wird ein Zyklus zu c_s hinzuaddiert.

Eingesparte Kosten: Eingesparte Kosten entstehen, wenn durch das Einfügen von Spillinstruktion Konflikte gelöst werden, die ursprünglich zwischen originalen Befehlen bestanden haben, und somit Stalls vermieden werden können. Die ursprünglichen Befehle werden jedoch nicht schneller abgearbeitet. Diese Einsparungen von Stalls werden in der Variable c_e aufaddiert und später bei der Verrechnung der jeweiligen Spillinstruktion gutgeschrieben. Wie bei den Stall-Kosten werden auch hier nur die Fälle 2 bis 5 betrachtet.

Zusätzliche Kosten: Zusätzliche Kosten können entstehen, wenn Parallelität verloren geht, die zuvor vorhanden war. Es gibt zwei hauptsächliche Ursachen, die zu diesem Verhalten führen.

- Eine Spillinstruktion, die zwischen zwei konfliktfreien IP-Befehlen mit dem selben Befehlscode eingefügt wurde, führt dazu, dass der zweite IP-Befehl nicht mehr innerhalb der Wiederholungsrate ausgeführt werden kann.
- Eine hinter einem IP-Befehl eingefügte Spillinstruktion sorgt dafür, dass ein nachfolgender LS-Befehl, der zuvor parallel mit dem IP-Befehl ausgeführt werden konnte, nun einen zusätzlichen Zyklus benötigt, um ausgeführt zu werden.

Es kann aber auch zu Fällen kommen, in denen sich die Verschiebungen gegenseitig aufheben. Angenommen wird wieder der Fall mit zwei IP-Befehlen, die den

selben Befehlscode besitzen und konfliktfrei sind. Wird hinter dem ersten eine Spillinstruktion eingefügt, die einen anderen Befehl nach hinten schiebt, kann dieser verschobene Befehl vielleicht nicht mehr parallel zum vorderen IP-Befehl ausgeführt werden und benötigt einen eigenen Zyklus. Als Konsequenz daraus kann der zweite IP-Befehl nicht mehr innerhalb der Wiederholungsrate ausgeführt werden, was wiederum dazu führt, dass ein zusätzlicher LS-Befehl parallel zu diesem ausgeführt werden könnte.

Mit Hilfe der vier zuvor beschriebenen Variablen c_d^a, c_s^a, c_e^a und c_z^a lassen sich die Kosten eines Basisblocks von den durch Spillinstruktionen erzeugten Kosten bereinigen. Die eingesparten Kosten stellen im eigentlichen Sinne einen Gewinn dar und werden deshalb in der Berechnung abgezogen. Insgesamt existieren zwei verschiedene Formulierungen, je nachdem, ob die Spillkosten hinzugerechnet oder abgezogen werden sollen. Liegen die $\mathrm{WCET}_{\mathrm{est}}$ -Werte der Basisblöcke vor, in die bereits Spillcode eingefügt wurde, so kann man die Kosten g_B des Basisblocks auf folgende Weise bereinigen, wobei a eine Spillinstruktion darstellt.

$$g_B^{\text{bereinigt}} = \mathrm{WCET}_{\mathrm{est}}^{\text{mit Spillcode}} - \sum_a c_d^a + c_s^a - c_e^a + c_z^a \tag{6.1}$$

Entsprechend lassen sich die Kosten eines Basisblocks mit neu eingefügtem Spillcode modellieren, wenn die bereinigten Werte bereit stehen.

$$g_B^{\text{mit Spillcode}} = g_B^{\text{bereinigt}} + \sum_a c_d^a + c_s^a - c_e^a + c_z^a \tag{6.2}$$

Eine besondere Bedeutung hat in beiden Verfahren der Begriff des Spillblocks. Deshalb soll dieser hier definiert werden, da er im Folgenden häufig Verwendung findet.

Definition 6.1 (Spillblock) *Ein Spillblock ist eine maximale Sequenz von Spillinstruktionen innerhalb des Programmcodes, deren Vorgänger- und Nachfolgebefehl – sofern vorhanden – keine Spillinstruktion ist. Die erste Spillinstruktion innerhalb dieser Reihe stellt den Kopf des Spillblocks dar.*

6.3 Integration in den WCC

Die Abbildung 6.1 zeigt eine Erweiterung des Aufbaus des Registerallokators aus Abschnitt 5.3. Hinzugekommen sind die beiden Pipelineanalyseverfahren. Man kann der Abbildung entnehmen, an welchen Stellen diese in die bestehende Struktur integriert werden. Die Kostenschätzung der Vorallokation kann durch den Ansatz der statischen Pipelineanalyse komplett ersetzt werden. Anders verhält es sich bei der dynamischen Pipelineanalyse. Diese stellt keine eigenständige Implementierung dar

und kann die bisherige ILP-Modellierung nicht ersetzen. Sie baut vielmehr auf dieser auf und erweitert sie um Variablen und Nebenbedingungen.

Grundsätzlich bedient sich das Verfahren der dynamischen Pipelineanalyse der ICD-LLIR als unterstützender, zentraler Datenstruktur. Diese wird als Hilfe zur Modellierung aller einzufügenden neuen Variablen und Nebenbedingungen benutzt. Einer Kopie der zu allokierenden ICD-LLIR werden dazu alle Spillinstruktionen hinzugefügt, die das Verfahren möglicherweise erzeugen kann (Abbildung 5.6 c und d). Damit kann für jede Spillinstruktion der Kontext untersucht werden, in dem sie eingefügt werden würde. Die statische Pipelineanalyse benutzt die zuvor schon vorhandene vorallokierte ICD-LLIR, die auch schon von den Kostenschätzverfahren benutzt wird.

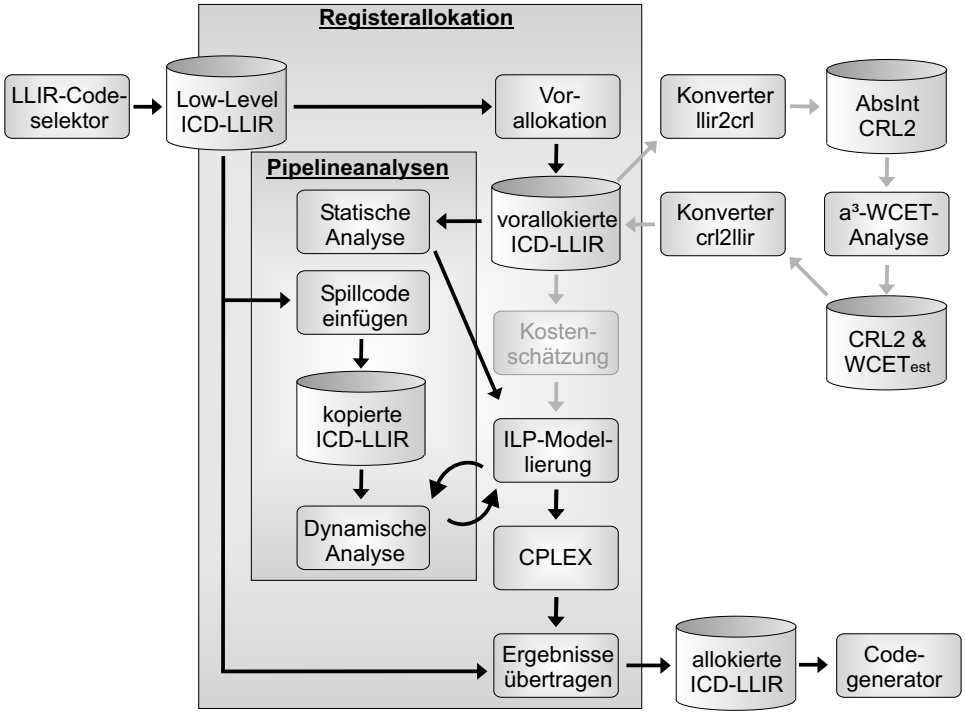


Abbildung 6.1: Integration der Pipelineanalysen in den Registerallokator

Die von Schmoll implementierten Verfahren fallen weder weg, noch werden sie unbrauchbar. Sie sind weiterhin über Kommandozeilenparameter des WCC ausführbar. Auf diese Weise können die verschiedenen Ansätze miteinander kombiniert werden. Die Kostenschätzverfahren oder die statische Pipelineanalyse können somit als Eingabe für Schmolls ILP-Modellierung oder die Erweiterung mit der dynamischen Pipelineanalyse verwendet werden. Wie die beiden neuen Knoten in Abbildung 6.1, die die Analyseverfahren repräsentieren, im Detail aufgebaut sind, wird in den folgenden Abschnitten gezeigt.

6.4 Statische Analyse der Vorallokation

Die statische Pipelineanalyse soll die bisher verwendeten Schätzverfahren ersetzen. Sie bestimmt für jeden Basisblock die Kosten nach der Vorallokation separat, die dieser ohne Spillcode hätte. Abbildung 6.2 zeigt schematisch den Ablauf des Verfahrens, das für jeden Basisblock unabhängig ausgeführt wird. Als Eingabe erhält es die vorallokierte ICD-LLIR und die von a^3 ermittelten WCET_{est}-Werte aller Basisblöcke, die jedoch durch Spillcode verursachte Kosten enthalten. In einem ersten Schritt werden innerhalb der ICD-LLIR alle Spillinstruktionen ausfindig gemacht. Im Anschluss geschieht die eigentliche Analyse, bei der für jede Spillinstruktion die vier Kostenvariablen c_d, c_s, c_e und c_z unabhängig voneinander ermittelt werden. Sobald diese vorliegen, können die Basisblockkosten mit Anwendung der Formel 6.1 bereinigt werden, und die Kosten des Basisblocks ohne eingefügten Spillcode stehen als neue WCET_{est} zur Verfügung.

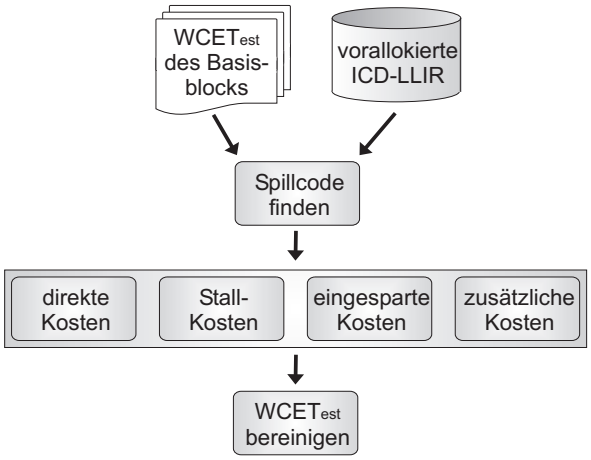


Abbildung 6.2: Schema der statischen Pipelineanalyse

Dieses Verfahren ist allgemein gehalten, weil es flexibel auf verschiedene Vorallokationen angewendet werden können soll. Diese wiederum fügen Spillcode auf sehr unterschiedliche Weise ein, weshalb alle Fälle berücksichtigt werden müssen. Eine Beschränkung auf ein Verfahren könnte die Implementierung teils stark vereinfachen, da Fälle unter Umständen wegfielen.

6.4.1 Implementierung

Wie die vier Kostenwerte c_d, c_s, c_e und c_z ermittelt werden, soll im Folgenden veranschaulicht und mit Beispielen unterlegt werden.

Direkte Kosten: Zur Bestimmung dieser Kosten c_d müssen unter Umständen mehrere Vorgänger der aktuell betrachteten Spillinstruktion untersucht werden. Die Be-

stimmung der Kosten kann relativ komplex werden. Folgende Eigenschaften müssen der Reihe nach beachtet werden:

1. Wo befindet sich der nächste IP-Befehl i vor der Spillinstruktion s, und befinden sich zwischen i und s ausschließlich LS-Befehle?
2. Ist s ein Speicherbefehl oder ein von i datenunabhängiger Ladebefehl?
3. Besitzt keiner der eventuell direkt vor s stehenden LS-Befehle eine Datenabhängigkeit zu i?
4. Ist die Latenz von i größer als Eins? (Wenn nicht, wird mit Punkt 6 fortgefahren.)
5. Benötigt i als Ausführungszeit die Latenz oder die Wiederholungsrate? Hierzu müssen wiederum folgende Fragen geprüft werden. Können alle drei Fragen mit ja beantwortet werden, so benötigt i nur die Wiederholungsrate.
   a) Befindet sich direkt vor i ein IP-Befehl i_v, der den selben Befehlscode besitzt und stehen zwischen i_v und i ausschließlich LS-Befehle?
   b) Benutzt i das Ergebnis von i_v auf keinen Fall als Faktor für seine Berechnung? (Anderenfalls würde ein Konflikt zwischen beiden Befehlen vorliegen, der eine gleichzeitige Ausführung der Execute-Stufe ausschließen würde.)
   c) Können alle LS-Befehle zwischen i_v und i parallel zu i_v ausgeführt werden? Ihre Anzahl darf daher die Ausführungszeit von i_v nicht überschreiten. Zusätzlich dürfen sie keine Datenabhängigkeit zu i_v besitzen, mit einer Ausnahme: Der letzte LS-Befehl darf ein datenabhängiger Speicherbefehl sein.
6. Ist die Anzahl der vor s direkt stehenden LS-Befehle kleiner als die Ausführungszeit von i?

Können die Fragen 1, 2, 3 und 6 mit ja beantwortet werden, so ist die Spillinstruktion parallel ausführbar und erzeugt keine Kosten. Diese Abfrage mutet sehr komplex an. Man muss an dieser Stelle aber beachten, dass nur Fälle derart komplex werden, bei denen IP-Befehle mit einer Latenz größer Eins vorhanden sind. Für eine Latenz vom Wert Eins ist die Bestimmung der Kosten c_d so einfach wie in Abbildung 4.4 und 4.5 beschrieben. Zur Veranschaulichung wird ein komplexeres Beispiel behandelt.

Der Ladebefehl L4 in Abbildung 6.3 soll in diesem Fall die eingefügte Spillinstruktion sein. Es stellt sich nun die Frage, ob L4 parallel zu M2 ausführbar ist. Dazu werden einfach die sechs Punkte der obigen Liste wie nachfolgend dargestellt abgearbeitet.

```
MADD.Q  d0,d0,d1,d2     ;M1
LD      d9,[a0]0        ;L1
LD      d10,[a1]0       ;L2
MADD.Q  d3,d3,d4,d5     ;M2
LD      d9,[a0]0        ;L3
LD      d10,[a1]0       ;L4
```

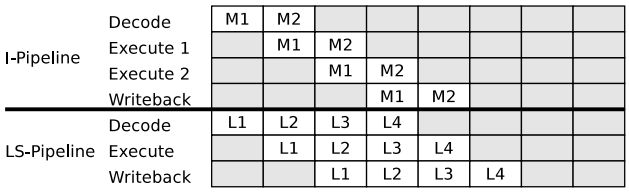


Abbildung 6.3: Beispiel für direkte Kosten einer Spillinstruktion

1.	ja	M2 ist ein solcher IP-Befehl.
2.	ja	L4 ist datenunabhängig von M2.
3.	ja	L3 ist ebenfalls datenunabhängig von M2.
4.	ja	Die Latenz von M2 beträgt Zwei.
5.		
a)	ja	M1 ist ein solcher IP-Befehl.
b)	ja	M1 und M2 sind konfliktfrei.
c)	ja	L1 und L2 können parallel zu M1 ausgeführt werden.
	3x ja	→ M2 benötigt einen Zyklus als Ausführungszeit.
6.	nein	Die Anzahl der LS-Befehle vor L4 ist nicht kleiner als die Ausführungszeit von M2.

Da die sechste Frage nicht mit ja beantwortet werden kann, ist eine Parallelausführung von L4 nicht möglich, was auch Abbildung 6.3 bereits zeigt.

Stall-Kosten: Um die Stall-Kosten c_s zu ermitteln, muss untersucht werden, ob die eingefügten Spillinstruktionen Konflikte mit den umgebenden Befehlen besitzen. Dazu wird überprüft, ob einer oder mehrere der in Tabelle 6.1 genannten Fälle 2 bis 5 eintreten. Für jeden Konflikt, der auftritt, wird ein Zyklus zu c_s hinzu addiert.

Beim zweiten Fall wird geprüft, ob nach einem eingefügten Ladebefehl ein LS-Befehl folgt, der das Register benutzt, welches der Ladebefehl definiert hat. Zu beachten ist, dass hier ein IP-Befehl zwischen den beiden Befehlen stehen kann.

Der dritte Fall bezieht sich auf Lade- und Speicherbefehle. Hier muss überprüft werden, ob bei einem eingefügten Ladebefehl im vorherigen Zyklus ein Speicherbefehl auf die selbe Adresse zugegriffen hat. Gleiches gilt für einen eingefügten Speicherbefehl, auf den ein Ladebefehl folgt. Sind beide Befehle, die einen solchen Konflikt besitzen, Spillinstruktionen, fallen die Kosten nur einmalig an.

Die Fälle 4 und 5 sind ziemlich trivial, da keinerlei Abhängigkeiten zu beachten sind. Der vierte tritt auf, wenn im Zyklus vor einer Spillinstruktion ein Funktionsaufruf stattfand, wobei vor dem LS-Befehl auch ein IP-Befehl stehen kann. Der fünfte Fall bezieht sich ausschließlich auf Speicherbefehle, denen direkt ein Rücksprungbefehl folgt.

Eingesparte Kosten: Durch das Einfügen von Spillinstruktionen können auch Konflikte unter den ursprünglichen Befehlen gelöst werden. Wiederum werden die

zuvor genannten Fälle 2 bis 5 aus Tabelle 6.1 betrachtet. Um solche Konflikte auflösen zu können, muss im Vorfeld irgendein LS-Befehl diesen verursacht haben. Die erzielten Einsparungen von Stalls werden in der Variable c_e aufaddiert.

In Abbildung 6.4 ist ein Beispiel für einen solchen Fall skizziert. Speicher- und Rücksprungbefehl würden normalerweise zu einem Stall führen, was dem fünften Fall aus Abschnitt 4.5 entspricht. Der Ladebefehl, der eine eingefügte Spillinstruktion sein soll, sorgt dafür, dass es zu keinem Stall kommt. Dadurch werden die beiden ursprünglichen Befehle nicht schneller abgearbeitet, aber die Variable c_e erhält den Wert Eins. Bei der Verrechnung heben sich c_d, welches den Wert Eins enthält, und c_e gegenseitig auf. Die Spillinstruktion erhält auf diese Weise Kosten Null, da diese auch in der Wirklichkeit keine Kosten erzeugt.

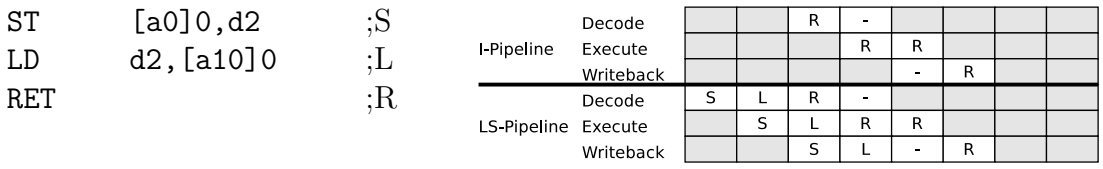


Abbildung 6.4: Beispiel für eingesparte Kosten durch Spillinstruktionen

Man kann hier gut erkennen, dass das Programm durch die eingesparten Kosten niemals schneller abgearbeitet werden kann, aber der Wert c_e wird für den Ladebefehl verrechnet und sorgt dafür, dass dieser insgesamt Kosten Null erhält.

Zusätzliche Kosten: Die Ermittlung der zusätzlichen Kosten ist ein komplexer Vorgang, ähnlich dem der Ermittlung der direkten Kosten. Zusätzliche Kosten entstehen durch verlorengegangene Parallelität, hervorgerufen durch eingefügten Spillcode. Im Prinzip werden alle LS-Befehle gesucht, die vor dem Einfügen des Spillcodes parallel zu einem IP-Befehl ausgeführt werden konnten, und für die dies im Nachhinein nicht mehr möglich ist. Ihre Anzahl entspricht der Höhe der zusätzlichen Kosten. Jeder dieser Befehle kann einer Spillinstruktion zugeordnet werden, die die verlorengegangene Parallelität hervorgerufen hat.

Im Prinzip wird eine vereinfachte Ablaufplanung implementiert. Diese bestimmt jeweils vor und nach dem Einfügen des Spillcodes, welche Ausführungszeit die Mehrzyklen-IP-Befehle haben und welche LS-Befehle parallel zu diesen ausgeführt werden können. Kann ein LS-Befehl nicht mehr parallel ausgeführt werden oder kann ein IP-Befehl nicht weiterhin die Wiederholungsrate ausnutzen, so werden die entstehenden Kosten der verursachenden Spillinstruktion zugeordnet.

In Abbildung 6.5 ist eine Erweiterung des Beispiels aus Abbildung 4.4 (unten) dargestellt. Der zusätzliche Speicherbefehl, der eine Spillinstruktion darstellen soll,

```
MADD.Q  d0,d0,d1,d2     ;M1
LD      d9,[a0]0        ;L1
LD      d10,[a1]0       ;L2
ST      [a2]0,d0        ;S
MADD.Q  d3,d3,d4,d5     ;M2
ADD     d6,d7,d8        ;A
```

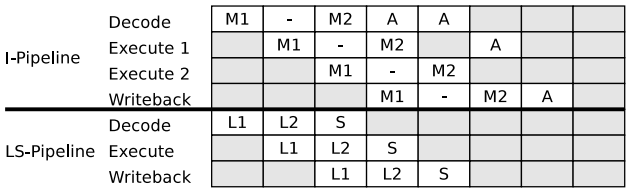


Abbildung 6.5: Beispiel für zusätzliche Kosten durch wegfallende Parallelität

erzeugt keinen Konflikt, aber er sorgt dafür, dass der zweite Befehl nicht die Wiederholungsrate ausnutzen kann, da beide IP-Befehle nicht mehr gleichzeitig in der Execute-Stufe ausgeführt werden können.

6.5 Dynamische Analyse innerhalb des ILP

Die dynamische Pipelineanalyse stellt kein eigenständiges Verfahren dar, es ist vielmehr eine Erweiterung der ILP-Formulierung von Schmoll. Während diese für alle Spillinstruktionen einen gleichen konstanten Kostenwert annimmt, modelliert die dynamische Pipelineanalyse die individuellen Kosten, die jede einzelne Spillinstruktion erzeugen würde. Da die Spillentscheidungen erst mit der Lösung des ILPs feststehen, und damit auch ihre Kosten, können diese nicht durch ein statisches Verfahren bestimmt werden, insbesondere, da die Spillentscheidungen sich gegenseitig in Bezug auf die entstehenden Kosten beeinflussen. Sie müssen daher innerhalb des ILPs mitmodelliert werden, um nach der Lösung den optimalen Wert annehmen zu können.

Das Schema der dynamischen Analyse (Abbildung 6.6) unterscheidet sich an einigen Stellen von dem der statischen Analyse (Abbildung 6.2). Der entscheidende Unterschied zwischen beiden Verfahren ist, dass im Gegensatz zur statischen bei der dynamischen Analyse zur Modellierungszeit nicht feststeht, welche Spillinstruktionen wirklich eingefügt werden und welche nicht. Außerdem werden die Arbeitsschritte bei der dynamischen Analyse für jede potenziell einzufügende Spillinstruktion separat ausgeführt.

Zuerst wird überprüft, ob die aktuelle Spillinstruktion die erste innerhalb des zugehörigen Spillblocks ist. In diesem Fall werden die eingesparten Kosten modelliert, die dieser Spillblock bei seiner Einfügung hervorruft, wobei nicht klar ist, ob überhaupt, und wenn ja welche Spillinstruktionen des Blocks eingefügt werden. Anschließend werden die übrigen Kosten modelliert. Die direkten und zusätzlichen Kosten werden für jede Spillinstruktion zusammen modelliert, allerdings unter der Einschränkung, dass eine mögliche Ausnutzung der Wiederholungsrate aufgrund der Komplexität nicht mitmodelliert wird. Unabhängig davon werden für jede Spillinstruktion die Stall-Kosten modelliert. Die erwähnten ILP-Erweiterungen werden im folgenden Abschnitt beschrieben.

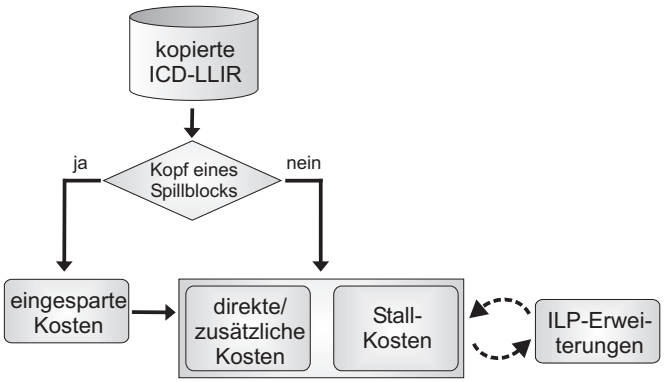


Abbildung 6.6: Schema der dynamischen Pipelineanalyse

Im Gegensatz zur statischen Analyse, die flexibel ist bzgl. anderer Vorallokatoren, werden in der dynamischen Analyse nur jene Fälle berücksichtigt, die wirklich durch das Einfügen von Spillcode an den durch die Abbildungen 5.6 (c) und (d) skizzierten Stellen entstehen können. An dieser Stelle soll darauf hingewiesen werden, dass die Begriffe der Spillinstruktion und der Spillentscheidung eine unterschiedliche Bedeutung besitzen. Eine Spillentscheidung (im Weiteren mit „E“ gekennzeichnet) repräsentiert eine zugehörige Spillinstruktion (im Weiteren mit „e“ gekennzeichnet) mit einem bestimmten physikalischen Register. Jede Spillinstruktion besitzt demnach mehrere zugehörige Spillentscheidungen, die alle verschiedene physikalische Register verwenden.

6.5.1 ILP-Erweiterungen

Bevor die Modellierung im Detail vorgestellt werden kann, werden noch einige ILP-Formulierungen benötigt. Insbesondere sind die Negation, Konkatenation und Disjunktionen für die verwendeten binären Entscheidungsvariablen des ILPs erforderlich. Die Formulierungen wurden [Kle08] entnommen und den Anforderungen entsprechend erweitert, da im Rahmen dieser Diplomarbeit Konjunktionen und Disjunktionen mehrerer Entscheidungsvariablen benötigt werden. Zunächst einmal soll die Negation definiert werden. Diese kann auf denkbar triviale Weise als Differenz modelliert werden. Die Entscheidungsvariable x enthält somit bei der Lösung des ILPs immer die Negation der Variable x_i.

$$x = \neg x_i \quad \longrightarrow \quad x = 1 - x_i \tag{6.3}$$

(6.4)

Die Konjunktion bzw. Disjunktion der Entscheidungsvariablen x_1 bis x_n kann wie folgt modelliert werden. Die Entscheidungsvariable x wird neu erstellt und erhält das Ergebnis der Konjunktion bzw. Disjunktion. Zusätzlich werden die jeweils rechts

aufgeführten Nebenbedingungen eingefügt, die die Korrektheit des Ergebnisses sicherstellen.

$$x = \bigwedge_{i=1}^{n} x_i \quad \longrightarrow \quad \begin{aligned} x &\geq \sum_{i=1}^{n} x_i \quad -(n-1) \\ x &\leq x_1 \\ &\vdots \\ x &\leq x_n \end{aligned} \tag{6.5}$$

$$x = \bigvee_{i=1}^{n} x_i \quad \longrightarrow \quad \begin{aligned} x &\leq \sum_{i=1}^{n} x_i \\ x &\geq x_1 \\ &\vdots \\ x &\geq x_n \end{aligned} \tag{6.6}$$

Zusätzlich werden einige Nebenbedingungen benötigt, auf denen die Modellierung der Kosten aufbaut. Zunächst wird die Entscheidungsvariable x_S^e definiert, die neu eingefügt wird. Sie nimmt den Wert Eins an, wenn eine der zu einer Spillinstruktion e gehörigen Spillentscheidungen den Wert Eins erhält und e somit eingefügt wird. Für einen Speicherbefehl müssen dazu alle Entscheidungsvariablen $x_{s\to p}^{st}$ aufsummiert werden (Formel 6.7), die das virtuelle Register s aus einem physikalischen Register p heraus sichern. Die Verwendung einer einfachen Summation genügt an dieser Stelle, weil die vorhandenen Nebenbedingungen nur zulassen, dass maximal eine der Variablen den Wert Eins annimmt. Analog kann für einen Ladebefehl eine solche Entscheidungsvariable (Formel 6.8) modelliert werden, indem alle zugehörigen Entscheidungsvariablen $x_{s\to p}^{ld}$ aufsummiert werden, die s in das Register p laden.

$$x_S^e = \sum_{p \in P} x_{s\to p}^{st} \tag{6.7}$$

$$x_S^e = \sum_{p \in P} x_{s\to p}^{ld} \tag{6.8}$$

Mit Hilfe der Entscheidungsvariable x_S^e können weitere Entscheidungsvariablen modelliert werden, deren Werte über die nachfolgenden Nebenbedingungen erzeugt werden. Die Spillinstruktion e soll an Position m innerhalb eines Spillblocks mit n Spillinstruktionen stehen. Die Entscheidungsvariable x_V^e nimmt den Wert Eins an, wenn mindestens eine der vorhergehenden Spillinstruktionen des Spillblocks vor e eingefügt wird. Die Disjunktion wird mittels der Transformation in Formel 6.6

erzeugt. Handelt es sich bei e um den Kopf eines Spillblocks, existieren folglich keine Spillinstruktionen direkt vor e und x_V erhält den Wert Null.

$$x_V^e = \bigvee_{i=1}^{m-1} x_S^i \tag{6.9}$$

Entsprechend gibt die Entscheidungsvariable x_N^e im Gegenzug an, ob mindestens eine der nachfolgenden Spillinstruktionen aus dem Spillblock hinter e eingefügt wird.

$$x_N^e = \bigvee_{i=m+1}^{n} x_S^i \tag{6.10}$$

Die Entscheidungsvariable x_B^e erhält den Wert Eins, wenn mindestens eine Spillinstruktion des zu e gehörigen Spillblocks eingefügt wird.

$$x_B^e = \bigvee_{i=1}^{n} x_S^i \tag{6.11}$$

Auch die Anzahl der direkt vor e eingefügten Spillinstruktionen wird benötigt und durch die Integer-Variable b_V^e modelliert.

$$b_V^e = \sum_{i=1}^{m-1} x_S^i \tag{6.12}$$

Neben den bereits vorgestellten Formulierungen wird auch eine ILP-Formulierung eines Größer-als benötigt, welches durch die nachfolgenden beiden Nebenbedingungen realisiert werden kann. Eine Integer-Variable b des ILPs soll mit einem konstanten Zahlenwert a verglichen werden. Die neu eingefügte Entscheidungsvariable $x_G^{b,a}$ soll den Wert Eins genau dann annehmen, wenn b größer als a ist, wobei $L = \infty$ ist. Die genaue Herleitung der Formulierung befindet sich in Anhang A.

$$\begin{aligned} x_G^{b,a} = b > a \quad \longrightarrow \quad & b \leq a + L x_G^{b,a} \\ & b \geq (a+1) - L(1 - x_G^{b,a}) \end{aligned} \tag{6.13}$$

Für die Übersichtlichkeit der folgenden Formulierungen werden zwei Platzhalter benötigt. Die Variable $x_{definiert}^{d,p}$ steht für alle Entscheidungsvariablen $x_{s\to p}^{def}$ und $x_{s\to p}^{ld}$, die einer Instruktion d zugeordnet sind, die das physikalische Register p definiert.

$$x_{definiert}^{d,p} = \bigvee_s x_{s\to p}^{def} \vee \bigvee_s x_{s\to p}^{ld} \qquad \text{, wobei } d \text{ das Register } p \text{ definiert} \tag{6.14}$$

Entsprechend steht die Variable $x^{d,p}_{benutzt}$ für alle Entscheidungsvariablen $x^{cont}_{s\to p}$ und $x^{ld}_{s\to p}$, die einer Instruktion d zugeordnet sind, die das physikalische Register p benutzt. Die Variable $x^{cont}_{s\to p}$ bzw. $x^{ld}_{s\to p}$ gibt dabei an, ob das virtuelle Register noch durch die letzte Benutzung einem physikalischen Register zugeordnet ist bzw. ob es unmittelbar vorher wieder aus dem Speicher geladen wurde.

$$x^{d,p}_{benutzt} = \bigvee_s x^{cont}_{s\to p} \vee \bigvee_s x^{ld}_{s\to p} \qquad \text{, wobei } d \text{ das Register } p \text{ benutzt} \tag{6.15}$$

Mit Hilfe dieser Platzhalter werden Entscheidungsvariablen modelliert, die jeweils den Wert Eins annehmen, wenn eine Datenabhängigkeit und damit ein Konflikt vorliegt. Unterschieden wird dabei zwischen den drei Datenabhängigkeiten WAW, RAW und WAR. Eine weitere Unterscheidung wird aufgrund der Art der Modellierung getroffen. Die Datenabhängigkeiten zwischen einer Spillentscheidung und einem Befehl oder zwischen zwei Befehlen müssen unterschiedlich modelliert werden. Für jede Spillentscheidung E, die das virtuelle Register s lädt oder sichert, ist das zugehörige physikalische Register p bekannt. Welche physikalischen Register den virtuellen Registern eines beliebigen Befehls zugeordnet werden, ist hingegen zur Modellierungszeit nicht bekannt, weshalb alle Fälle modelliert werden müssen.

Um einen Konflikt zwischen E, das zu einer Spillinstruktion e gehört, und einer anderen beliebigen Instruktion d zu finden, muss überprüft werden, ob die Instruktion die selben physikalischen Register wie die Spillentscheidung benutzt bzw. definiert. Dieser einfache Abgleich kann durch Entscheidungsvariablen und Nebenbedingungen in den Formeln 6.16 und 6.17 modelliert werden.

$$x^{E,d,p}_{WAW} = x^{d,E,p}_{RAW} = \bigvee_s x^{d,p}_{definiert} \tag{6.16}$$

Die Variablen $x^{E,d,p}_{WAW}$ bzw. $x^{d,E,p}_{RAW}$ nehmen den Wert Eins an, wenn zwischen einer Spillentscheidung E und dem Befehl d ein WAW- bzw. RAW-Konflikt vorliegt, wobei sich d im Programmcode vor e befindet. Die Möglichkeit, beide Fälle auf dieselbe Weise modellieren zu können, liegt darin begründet, dass das durch die Spillentscheidung definierte bzw. benutzte Register bekannt ist. Die durch den Befehl d definierten Register werden über den Platzhalter $x^{d,p}_{definiert}$ berücksichtigt, der in Formel 6.14 eingeführt wird.

$$x^{E,d,p}_{RAW} = \bigvee_s x^{d,p}_{benutzt} \tag{6.17}$$

Analog nimmt die Variable $x^{E,d,p}_{RAW}$ den Wert Eins an, wenn zwischen einer Spillentscheidung E und dem Befehl d ein RAW-Konflikt vorliegt, wobei sich e im Programmcode vor d befindet. Die von dem Befehl d benutzten Register werden durch den Platzhalter $x^{d,p}_{benutzt}$ aus Formel 6.15 berücksichtigt.

Komplizierter hingegen wird es, wenn überprüft werden soll, ob zwei beliebige Instruktionen d und e datenabhängig sind. An dieser Stelle müssen alle Kombinationen von Entscheidungsvariablen und physikalischen Registern P berücksichtigt werden. Die entstehenden Konflikte sollen daher komplexe Konflikte heißen. Die Entscheidungsvariable auf der linken Seite der Nebenbedingungen 6.18 und 6.19 nimmt jeweils den Wert Eins an, wenn ein Konflikt zwischen beiden Befehlen auftritt.

$$x^{d,e}_{RAW_{komplex}} = \bigvee_{p \in P} ((\bigvee_{s} x^{d,p}_{definiert}) \wedge (\bigvee_{s} x^{e,p}_{benutzt})) \tag{6.18}$$

$$x^{d,e}_{WAW_{komplex}} = \bigvee_{p \in P} ((\bigvee_{s} x^{d,p}_{definiert}) \wedge (\bigvee_{s} x^{e,p}_{definiert})) \tag{6.19}$$

Zur Vereinfachung der folgenden Formulierungen wird an dieser Stelle noch eine Spezialisierung der beiden zuvor genannten Formeln eingeführt, die nur der Anschaulichkeit dient. Innerhalb der ILP-Formulierung werden ausschließlich komplexe Konflikte zwischen einem IP-Befehl d und einer nachfolgenden Spillinstruktion e modelliert. Insbesondere wird für einen Speicher- bzw. Ladebefehl nur die Entscheidungsvariable $x^{d,e}_{RAW_{komplex}}$ bzw. $x^{d,e}_{WAW_{komplex}}$ eingefügt.

$$x^{d,e}_{komplex} = \begin{cases} x^{d,e}_{RAW_{komplex}} & \text{wenn } e \text{ ein Speicherbefehl ist} \\ x^{d,e}_{WAW_{komplex}} & \text{wenn } e \text{ ein Ladebefehl ist} \end{cases} \tag{6.20}$$

6.5.2 Modellierung

Mit Hilfe der kopierten ICD-LLIR kann für jede Spillentscheidung der Kontext untersucht werden, in dem sie eingefügt wird. Zu beachten bleibt dabei insbesondere, dass Spillentscheidungen von der Einfügung anderer Spillinstruktionen beeinflusst werden können. Für diese steht nicht fest, ob sie eingefügt werden oder nicht. Diese Variabilität muss mit Hilfe der zu diesen Spillinstruktionen zugehörigen Entscheidungsvariablen modelliert werden.

Die direkten und zusätzlichen Kosten jeder Spillentscheidung E werden wie zuvor erwähnt kombiniert modelliert und hier durch die ganzzahlige Variable c^E_k repräsentiert. Nicht berücksichtigt werden alle Effekte, die sich auf die Wiederholungsrate beziehen. Dies würde die Komplexität des erzeugten ILPs drastisch steigern, da unter Umständen weitreichende Abhängigkeiten modelliert werden müssten. Außerdem ist die Anzahl der Fälle, in denen die Wiederholungsrate ausgenutzt werden kann, verschwindend gering, so dass sich eine Implementierung aufgrund der Komplexität rein praktisch nicht lohnt. Die Stall-Kosten c^E_s werden separat für alle Spillinstruktionen modelliert. Handelt es sich bei der aktuell betrachteten Spillinstruktion um den Kopf eines Spillblocks, so werden an dieser Stelle die zum Spillblock zugehörigen eingesparten Kosten c^{sb}_e modelliert, die entstehen, wenn Spillinstruktionen dieses

Blocks Konflikte lösen. Ausgehend von den Formeln 6.2 und 5.1 wird folgende Formel für die Basisblockkosten d_B modelliert.

$$d_B = g_B^{\text{bereinigt}} + \sum_E (c_k^E + c_s^E) - \sum_{sb} c_e^{sb} \tag{6.21}$$

Die Variable $g_B^{\text{bereinigt}}$ entspricht den bereinigten Kosten des jeweiligen Basisblocks, wie sie durch die statische Pipelineanalyse oder die Kostenschätzverfahren aus Abschnitt 5.2.4 zur Verfügung gestellt werden. Die Modellierung der drei auftretenden Kostenvariablen wird im Folgenden detailliert vorgestellt.

Kombination der direkten und zusätzlichen Kosten c_k^E: Nachfolgend werden die modellierten Kosten für eingefügte Speicher- und Ladebefehle separat aufgeführt, da Speicherinstruktionen auch parallel zu einem vorhergehenden IP-Befehl ausgeführt werden können, wenn sie eine Datenabhängigkeit zu diesem besitzen, was für Ladebefehle nicht gilt. Deswegen entspricht die Variable c_k^E im Folgenden den Variablen c_k^{st} im Fall von Speicherbefehlen und c_k^{ld} bei Ladebefehlen.

Für einige Spillentscheidungen kann teilweise schon zur Modellierungszeit festgestellt werden, ob sie mit Sicherheit einen Konflikt verursachen, weil durch Vorfärbungen einigen Befehlen bestimmte physikalische Register fest zugeordnet werden und zu jeder Spillentscheidung das physikalische Register ohnehin bekannt ist. Dieses Wissen über sichere Konflikte kann direkt in die Modellierung übernommen werden. Den entsprechenden Spillentscheidungen werden feste Kosten von Eins zugewiesen. Ebenso kann für einige Spillentscheidungen ein Konflikt generell ausgeschlossen werden. Alle übrigen Spillentscheidungen können möglicherweise einen Konflikt auslösen, wenn die Allokation die virtuellen Register der Befehle den entsprechenden physikalischen Registern zuordnet. Diese möglichen Konflikte, die dem ersten und sechsten Fall aus Tabelle 6.1 entsprechen, werden über die Formeln 6.16 bis 6.20 modelliert.

Zusätzlich wird noch eine weitere Begrifflichkeit benötigt. Grundsätzlich können zu einem IP-Befehl eine Anzahl von LS-Befehlen parallel ausgeführt werden. Diese Anzahl entspricht der Latenz aus Tabelle 4.1. Der Begriff der Zeitnischen soll angeben, wie viele Plätze für Spillinstruktionen zur Verfügung stehen, um parallel mit dem IP-Befehl ausgeführt zu werden. Dazu wird die Anzahl der ursprünglichen LS-Befehle bestimmt, die hinter dem IP-Befehl standen und parallel mit diesem ausgeführt werden konnten. Die Anzahl der freien Zeitnischen entspricht der Differenz aus der Latenz und dieser Anzahl. Dementsprechend werden die zusätzlichen Kosten direkt mitmodelliert. Beispielsweise besitzt ein IP-Befehl mit Latenz Eins, auf den ursprünglich keine LS-Befehle folgten, eine freie Zeitnische zur parallelen Ausführung von Spillinstruktionen.

Jede Spillentscheidung E modelliert die Zuordnung für ein bestimmtes physikalisches Register p. Um die einzelnen Kostenmodellierungen für alle Spillentscheidungen zu erhalten, wird eine Fallunterscheidung vorgenommen. Die einzelnen Fälle

hängen vom Kontext ab, in dem die zu modellierende Spillentscheidung eingefügt werden soll. Dabei werden für die aktuelle Spillentscheidung beim ersten beginnend alle Fälle nacheinander überprüft. Bei einer Übereinstimmung wird die entsprechende Modellierung übernommen und die nachfolgenden Fälle werden nicht mehr berücksichtigt.

1. In diesem Fall wird der Spillentscheidung die ursprüngliche Entscheidungsvariable zugeordnet, um feste Kosten von Eins zu modellieren. Eine solche Modellierung kann verschiedene Ursachen haben:
   a) Vor der betrachteten Spillinstruktion befindet sich kein IP-Befehl, zu dem diese parallel ausgeführt werden könnte.
   b) Die entsprechende Spillentscheidung verursacht einen sicheren WAW-Konflikt zu dem IP-Befehl, mit die zugehörige Spillinstruktion parallel ausgeführt werden könnte.
   c) Der IP-Befehl, mit dem die zugehörige Spillinstruktion parallel ausgeführt werden könnte, besitzt keine freien Zeitnischen mehr.

$$c_k^{st} = x_{s \to p}^{st} \tag{6.22}$$
$$c_k^{ld} = x_{s \to p}^{ld} \tag{6.23}$$

2. Die Spillinstruktion e folgt direkt auf einen IP-Befehl d. Das heißt, dass vor der Spillinstruktion keine andere Spillinstruktion eingefügt werden kann.

$$c_k^{st} = 0 \tag{6.24}$$
$$c_k^{ld} = x_{WAW_e}^{E,d,p} \wedge x_{s \to p}^{ld} \tag{6.25}$$

   Spillspeicherbefehle sind in diesem Fall immer parallel ausführbar und erzeugen keine Kosten. Bei Spillladebefehlen muss an dieser Stelle zusätzlich eine mögliche Datenabhängigkeit über die Entscheidungsvariable $x_{WAW_e}^{E,d,p}$ berücksichtigt werden. Kann ein solcher Konflikt während der Modellierung jedoch sicher ausgeschlossen werden, so kann sogar der rechte Teil der Formel 6.25 durch 0 ersetzt werden.

3. Zwischen der zu einer Spillentscheidung E gehörigen Spillinstruktion e und einem davor stehenden IP-Befehl d stehen ausschließlich andere Spillinstruktionen. Zudem besitzt d nur eine freie Zeitnische, weshalb nur ein LS-Befehl parallel ausgeführt werden kann.

$$c_k^{st} = x_V^{st} \wedge x_{s \to p}^{st} \tag{6.26}$$
$$c_k^{ld} = (x_V^{ld} \vee x_{WAW_e}^{E,d,p}) \wedge x_{s \to p}^{ld} \tag{6.27}$$

Müssen mehrere Spillinstruktionen beachtet werden, kann es sein, dass ein Spillspeicherbefehl nicht mehr parallel zu d ausgeführt werden kann. Daher muss zusätzlich mit Hilfe der Entscheidungsvariable x_V^{st} modelliert werden, ob eine der vorherstehenden Spillinstruktionen eingefügt wurde. Diese Erweiterung betrifft auch die Spillladebefehle.

4. Zwischen der zu einer Spillentscheidung E gehörigen Spillinstruktion e und einem davor stehenden IP-Befehl d stehen ausschließlich andere Spillinstruktionen. Zudem ist deren Anzahl kleiner als die Anzahl der freien Zeitnischen von d. Demnach könnte die Spillinstruktion parallel zu d ausgeführt werden, wenn die vorherstehenden Spillinstruktionen F keinen Konflikt verursachen.

$$c_k^{st} = \bigvee_{f \in F} x_{komplex}^{d,f} \wedge x_{s \to p}^{st} \tag{6.28}$$

$$c_k^{ld} = (x_{WAW_e}^{E,d,p} \vee (\bigvee_{f \in F} x_{komplex}^{d,f})) \wedge x_{s \to p}^{ld} \tag{6.29}$$

An dieser Stelle müssen zusätzlich die komplexen Konflikte $x_{komplex}^{d,f}$ der vorherstehenden Spillinstruktionen F beachtet werden, die diese zu d haben können. An dieser Stelle können beide rechten Seiten der Nebenbedingungen sogar durch 0 ersetzt werden, wenn alle Konflikte während der Modellierung ausgeschlossen werden können. Die Spillentscheidung würde dementsprechend nie direkte Kosten verursachen.

5. In allen anderen Fällen kann die zu einer Spillentscheidung E gehörige Spillinstruktion e nur parallel zum IP-Befehl d ausgeführt werden, wenn entsprechend viele vorhergehende Spillinstruktionen nicht eingefügt werden und alle eingefügten, vorhergehenden Spillinstruktionen F keinen Konflikt verursachen. Des Weiteren muss d ausreichend viele freie Zeitnischen z zur Verfügung stellen, und E selber darf keine Datenabhängigkeit zu d besitzen, wenn e ein Ladebefehl ist.

$$c_k^{st} = (x_G^{b_V^e,(z-1)} \vee (\bigvee_{f \in F} x_{komplex}^{d,f})) \wedge x_{s \to p}^{st} \tag{6.30}$$

$$c_k^{ld} = (x_{WAW_e}^{E,d,p} \vee x_G^{b_V^e,(z-1)} \vee (\bigvee_{f \in F} x_{komplex}^{d,f})) \wedge x_{s \to p}^{ld} \tag{6.31}$$

In diesen Formulierungen wird zusätzlich das Größer-als aus Formel 6.13 verwendet, um zu modellieren, dass auch wirklich ausreichend freie Zeitnischen z zur Verfügung stehen.

Stall-Kosten c_s^E: Die Stall-Kosten entstehen durch Addition von vier einzelnen Kostenvariablen, die die Fälle 2 bis 5 aus Tabelle 6.1 repräsentieren. Diese vier

Kostenvariablen sind allesamt Entscheidungsvariablen, da jeder Fall Kosten von 1 verursacht.

$$c_s^E = c_{Fall\,2} + c_{Fall\,3} + c_{Fall\,4} + c_{Fall\,5} \tag{6.32}$$

Im Folgenden werden die Modellierungen der vier Einzelfälle beschrieben, wobei für jede Spillentscheidung auf Grundlage des zugehörigen physikalischen Registers zur Modellierungszeit bestimmt werden kann, ob ein Konflikt ausgeschlossen werden kann oder ob ein Konflikt sicher oder möglicherweise vorliegt. Bei allen Fällen muss beachtet werden, dass zwischen den erwähnten Instruktionen noch weitere Spillinstruktionen eingefügt werden können. Diese werden über die Variablen x_V^e und x_N^e berücksichtigt. Alle Fälle enthalten zusätzlich die jeweilige zur Spillentscheidung gehörige Entscheidungsvariable $x_{s\to p}^{ld}$ bzw. $x_{s\to p}^{st}$, um sicherzustellen, dass die Kosten nur dann anfallen, wenn die Entscheidungsvariable den Wert Eins erhält.

2. Bei diesem Fall sind zwei Unterfälle denkbar, die entstehen könnten.

   a) Es liegt eine Spillentscheidung E für einen Spillladebefehl e vor, die ein Adressregister p definiert, und auf die ein Befehl f folgt, der keine Spillinstruktion ist und p benutzt.

   $$c_{Fall\,2} = \begin{cases} 0 & \text{kein Konflikt} \\ \neg x_N^e \wedge x_{s\to p}^{ld} & \text{sicherer Konflikt} \\ \neg x_N^e \wedge x_{s\to p}^{ld} \wedge x_{RAW_e}^{E,f,p} & \text{möglicher Konflikt} \end{cases} \tag{6.33}$$

   Um diesen Konflikt zu modellieren, wird über die Variable $\neg x_N^e$ sichergestellt, dass keine Spillinstruktion direkt nach e eingefügt wird. Ein möglicher Konflikt zu f wird über die Variable $x_{RAW_e}^{E,f,p}$ modelliert. Kann eine Datenabhängigkeit von vornherein ausgeschlossen werden, so liegen die Kosten bei Null.

   b) Es liegt eine Spillentscheidung E für einen Spillspeicherbefehl e vor, die ein Adressregister p benutzt, und vor der ein Ladebefehl d steht, der p definiert.

   $$c_{Fall\,2} = \begin{cases} 0 & \text{kein Konflikt} \\ \neg x_V^e \wedge x_{s\to p}^{st} & \text{sicherer Konflikt} \\ \neg x_V^e \wedge x_{s\to p}^{st} \wedge x_{RAW_e}^{E,d,p} & \text{möglicher Konflikt} \end{cases} \tag{6.34}$$

   Die Variable $\neg x_V^e$ stellt sicher, dass keine Spillinstruktion direkt vor e eingefügt wird. Ein möglicher Konflikt zu d wird über die Variable $x_{RAW_e}^{E,d,p}$ modelliert. Auch hier liegen die Kosten bei Null, wenn eine Datenabhängigkeit sicher ausgeschlossen werden kann.

3. Beim dritten Fall sind vier Unterfälle denkbar, von denen drei jedoch ausgeschlossen werden können.

a) Es liegen eine Spillladeinstruktion e, mit der zugehörigen Spillentscheidung E, und eine davor stehende Spillspeicherinstruktion f vor, die beide auf die selbe Adresse im Speicher zugreifen. Der Zugriff auf dieselbe Speicherstelle ist dann der Fall, wenn beide Spillinstruktionen das selbe virtuelle Register s verwenden. Zwischen diesen beiden Instruktionen können weitere Spillinstruktionen stehen. Ein solches Szenario ist denkbar, wird aber von der Registerallokation nicht erzeugt, da dies zusätzliche Kosten erzeugen würde, was die Zielfunktion verhindert.

b) Ein Szenario wie in a) ist auch mit einem IP-Befehl d zwischen den beiden Spillinstruktionen denkbar, wobei hinter dem Speicher- und vor dem Ladebefehl weitere Spillinstruktionen stehen können, die nicht eingefügt werden. In diesem Fall darf d das Register s nicht definieren, da der Fall ansonsten ausgeschlossen werden kann.

$$c_{Fall\,3} = \neg x_N^f \wedge x_S^f \wedge \neg x_V^e \wedge x_{s\rightarrow p}^{ld} \tag{6.35}$$

Die Variable $\neg x_N^f$ bzw. $\neg x_V^e$ stellt sicher, dass keine Spillinstruktion direkt nach f bzw. vor e eingefügt wird. Zusätzlich wird über die Variable x_S^f die Bedingung modelliert, dass f überhaupt eingefügt ist, da der Konflikt sonst nicht entstehen kann. Ein möglicher Konflikt zwischen dem IP- und Ladebefehl muss an dieser Stelle nicht modelliert werden, weil er durch die Betrachtung der verwendeten virtuellen Register zur Modellierungszeit erkannt wird.

c) Beim dritten Unterfall folgt auf eine Spillladeinstruktion ein Speicherbefehl, der keine Spillinstruktion ist, und beide Befehle greifen auf dieselbe Adresse im Speicher zu. Dieser Fall kann ausgeschlossen werden, weil Spillinstruktionen und die übrigen LS-Instruktionen auf verschiedene Adressbereiche des Speichers zugreifen und somit niemals die gleiche Speicherstelle ansprechen können.

d) Der vierte Unterfall entspricht Unterfall c), mit dem Unterschied, dass hier noch ein IP-Befehl zwischen den Befehlen steht. Auch dieser Fall kann aus dem zuvor genannten Grund ausgeschlossen werden.

4. Der vierte Fall macht zunächst eine Unterscheidung notwendig. Ein Befehl wird nach einem Rücksprungbefehl ausgeführt, wenn direkt vor ihm ein Funktionsaufruf steht. Andererseits wird ein Befehl nach einem Funktionsaufruf ausgeführt, wenn er der erste in der Funktion ist. Wird eine solche Spillinstruktion gefunden, die einer der beiden Bedingungen genügt, so sind für diesen Fall noch zwei Unterfälle vorstellbar, wobei jeweils zwischen Speicher- und Ladebefehl unterschieden werden muss.

   a) Die Spillinstruktion e folgt direkt auf einen Funktionsaufruf oder Rücksprungbefehl.

für Speicherbefehle:

$$c_{Fall\,4} = \neg x_V^e \wedge x_{s\to p}^{st} \tag{6.36}$$

für Ladebefehle:

$$c_{Fall\,4} = \neg x_V^e \wedge x_{s\to p}^{ld} \tag{6.37}$$

Die Variable $\neg x_V^e$ stellt sicher, dass keine Spillinstruktion zwischen dem Funktionsaufruf bzw. Rücksprungbefehl und e liegt.

b) Zwischen der Spillinstruktion e mit der zugehörigen Spillentscheidung E und dem Funktionsaufruf oder Rücksprungbefehl steht ein IP-Befehl d, zu dem e parallel ausgeführt werden kann. Deshalb muss beachtet werden, dass sowohl vor e als auch vor d mögliche Spillinstruktionen sein können, die jedoch nicht eingefügt werden. Die Spillinstruktion f soll der Kopf des vor d möglichen Spillblocks sein.

für Speicherbefehle:

$$c_{Fall\,4} = \neg(x_V^e \vee x_B^f) \wedge x_{s\to p}^{st} \tag{6.38}$$

für Ladebefehle:

$$c_{Fall\,4} = \begin{cases} \neg(x_V^e \vee x_B^f) \wedge x_{s\to p}^{ld} & \text{kein Konflikt} \\ 0 & \text{sicherer Konflikt} \\ \neg(x_V^e \vee x_B^f) \wedge x_{WAW_e}^{E,d,p} \wedge x_{s\to p}^{ld} & \text{möglicher Konflikt} \end{cases} \tag{6.39}$$

Hier stellt die Variable x_V^e bzw. x_B^f sicher, dass sich keine Spillinstruktion direkt vor e bzw. vor d befindet. Einen möglichen Konflikt zwischen d und einem Spillladebefehl e modelliert die Variable $x_{WAW_e}^{E,d,p}$. Liegt eine solche Datenabhängigkeit sicher vor, kann der Konflikt nie auftreten.

5. Beim fünften Fall muss lediglich überprüft werden, ob nach einer Spillspeicherinstruktion e ein Rücksprungbefehl folgt, wobei zwischen beiden Befehlen ausschließlich Spillinstruktionen stehen dürfen. Es muss nun noch über die Variable $\neg x_N^e$ modelliert werden, dass keine von diesen eingefügt wird.

$$c_{Fall\,5} = \neg x_N^e \wedge x_{s\to p}^{st} \tag{6.40}$$

Eingesparte Kosten c_e^{sb}: Die eingesparten Kosten beziehen sich immer auf einen kompletten Spillblock. Wird eine Spillinstruktion des Blocks eingefügt, können Konflikte zwischen den ursprünglich vorhandenen Instruktionen gelöst werden. Die Höhe der eingesparten Kosten a eines Spillblocks wird genauso ermittelt, wie dies in Abschnitt 6.4.1 für einzelne Spillinstruktionen geschieht. In der Modellierung wird über

die Variable x_B^e sichergestellt, dass mindestens eine der Spillinstruktionen des Spillblocks den Wert Eins annimmt, wobei die Spillinstruktion e der Kopf des Spillblocks sein soll.

$$c_e^{sb} = a \cdot x_B^e \tag{6.41}$$

„Die Überführung des Optimierungsproblems der Registerallokation in ein ILP erzeugt eine strukturierte Beschreibung des Problems. Außerdem können einem ILP leicht zusätzliche Anforderungen hinzugefügt werden.“

Florian Schmoll
[Sch08, Seite 3]

7 Erweiterungen der Registerallokation

In diesem Kapitel werden mehrere Erweiterungen beschrieben, die an dem bestehenden Registerallokationsverfahren von Schmoll im Rahmen dieser Arbeit vorgenommen werden. Diese sind alle voneinander und von den in Kapitel 6 vorgestellten Verfahren unabhängig. Ihre Berücksichtigung beruht entweder auf den Verbesserungsvorschlägen aus Abschnitt 5.4 oder auf eigenen Beobachtungen, die im Laufe der Bearbeitung gemacht wurden. Die Abbildung 7.1 zeigt, an welchen Stellen die einzelnen Ansätze, die in den folgenden Abschnitten beschrieben werden, innerhalb des Registerallokators eingefügt werden. Alle Ansätze haben das Ziel, die Güte der Registerallokation bzgl. der WCET zu verbessern.

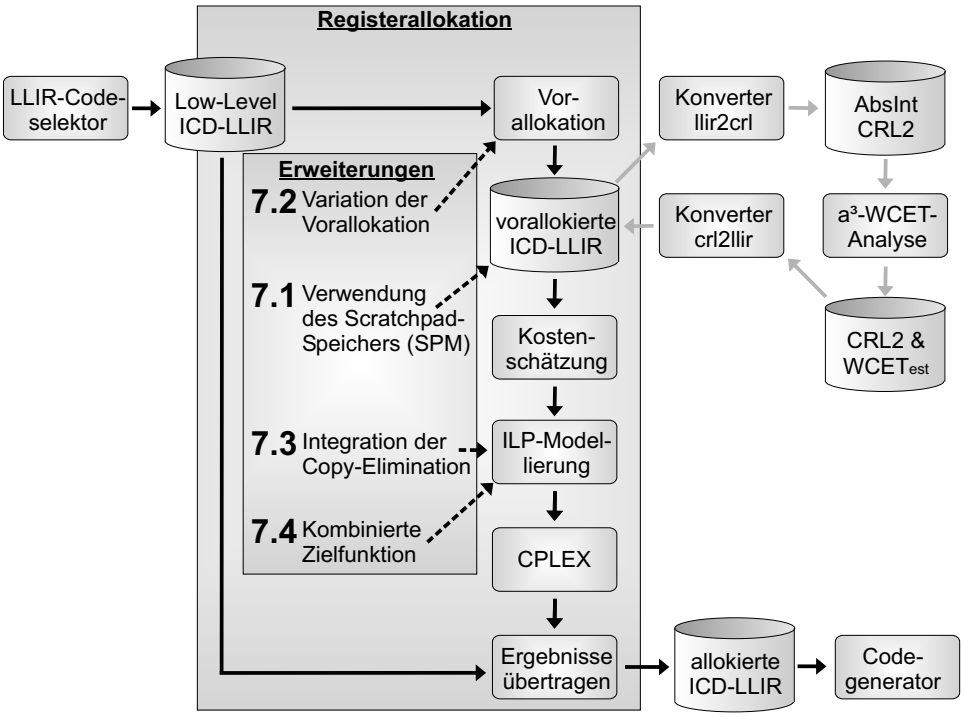


Abbildung 7.1: Erweiterungen der Registerallokation

7.1 Verwendung des Scratchpad-Speichers (SPM)

Im Laufe der Arbeit hat sich herausgestellt, dass es sich negativ auf die WCET-Analyse der Vorallokation auswirkt, wenn der Programmcode im Flash-Speicher der PMU abgelegt wird. Dies rührt daher, dass für den Flash ein Paging eingesetzt wird [Inf07]. Will die Fetch-Stufe die folgenden Befehle lesen, muss sich der Flash-Speicher im Lesemodus befinden und die jeweilige Seite, die aus 64 Wörtern (256 Bytes) besteht, geladen sein. Die CPU kann dann schnell auf die einzelnen Stellen zugreifen. Für das Laden einer anderen Seite oder Änderungen des Speicherinhalts muss der Flash-Speicher in den Kommandomodus wechseln. Die einzelnen Operationen, die hier ausgeführt werden, benötigen i. d. R. mehr als einen Zyklus, teilweise sogar bis zu sechs Zyklen. Somit treten Seiteneffekte auf, die die $\mathrm{WCET}_{\mathrm{est}}$ der einzelnen Basisblöcke beeinflussen.

Beispielsweise besitzt ein Basisblock, der innerhalb einer Schleife mehrfach ausgeführt wird, mehrere Ausführungskontexte, die auch den Zustand des Flash-Speichers umfassen. In einer oder mehreren Ausführungen des Basisblocks kann es dazu kommen, dass verschiedene Flash-Operationen ausgeführt werden müssen und die durch das Analyseprogramm a^3 ermittelte $\mathrm{WCET}_{\mathrm{est}}$ dieser Ausführung stark ansteigt. Für andere Ausführungen muss das nicht der Fall sein. Da in der Modellierung des WCEP immer die größte $\mathrm{WCET}_{\mathrm{est}}$ des Basisblocks mit dessen Ausführungsanzahl multipliziert wird, kann es zu Verzerrungen innerhalb der Modellierung kommen. Eine weitere und noch nachteiligere Eigenschaft des Flash-Speichers ist, dass das Einfügen von Spillcode nicht nur die Kosten des aktuellen Basisblocks verändert, sondern auch die Kosten der anderen Basisblöcke. Die beobachteten Verschiebungen sind dabei teils schwerwiegend.

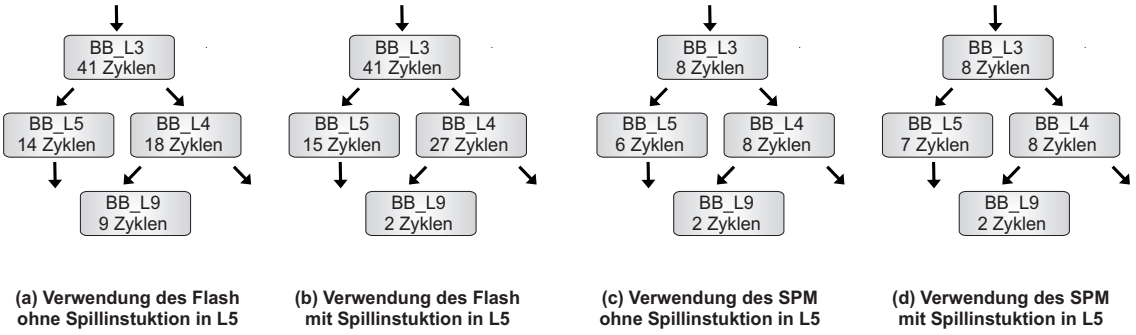


Abbildung 7.2: Auswirkung des verwendeten Speichers auf die WCET_{est}

Die Abbildung 7.2 zeigt einen Ausschnitt des zum Benchmark *binarysearch* (Anhang B) gehörigen CFG. Fall (a) zeigt die $\mathrm{WCET}_{\mathrm{est}}$-Werte die aus der WCET-Analyse der Vorallokation stammen, wenn der Programmcode im Flash-Speicher liegt. Im Gegensatz zu (a) wurde in (b) eine zusätzliche Spillinstruktion in den Basisblock L5 eingefügt. Man kann erkennen, dass sich nicht nur die Werte für L5 sondern auch die Werte für die anderen Basisblöcke ändern. Die Fälle (c) und (d)

stellen die selbe Situation unter Verwendung des SPM dar. Hier wird deutlich, dass sich das Einfügen der Spillinstruktion ausschließlich auf den entsprechenden Basisblock auswirkt. Zudem ist ersichtlich, dass die WCET_{est}-Werte bei Verwendung des SPM durchweg niedriger liegen, da die Effekte des Flash-Speichers entfallen.

Um diese Eigenheiten des Flash-Speichers auszuschließen, wird dem Analyseprogramm a^3 die Einstellung übergeben, dass der zu analysierende Programmcode dem SPM innerhalb PMI zugeordnet sein soll. Dies beeinflusst nicht die endgültige Positionierung des Programmcodes, da diese Einstellung ausschließlich die Analyse betrifft. Die Verschiebung des Programmcodes in den SPM bewirkt des Weiteren keinerlei Änderungen bzgl. der Daten. Diese verbleiben in den vorgesehenen Speichern.

Der SPM hat die Eigenschaft, dass Speicherzugriffe immer Latenz Eins besitzen und somit innerhalb eines Zyklus abgearbeitet werden können. Die benötigten Befehle stehen der Fetch-Stufe also unmittelbar zur Verfügung. Es hat sich gezeigt, dass die WCET_{est}-Werte der einzelnen Ausführungen nur in geringem Maße voneinander abweichen. Dies lässt sich nicht verhindern, weil aufgrund von Parallelausführungen die Kosten mancher Instruktionen teilweise entfallen. Zudem korrelieren die WCET_{est}-Werte jedes Basisblocks deutlich stärker mit den enthaltenen Befehlen und deren Ausführungszeiten, was vorher durch die Verschiebungen oft nicht der Fall war. Des Weiteren beeinflusst das Einfügen einer Spillinstruktion praktisch nur die Kosten des zugehörigen Basisblocks. Geringfügige Verschiebungen aufgrund von Parallelausführungen hin zum Vorgänger- oder Nachfolgerbasisblock sind dabei zu vernachlässigen.

7.2 Variation der Vorallokation

Die bisherige Implementierung des Registerallokators verwendet die naive Registerallokation RASA des WCC als Vorallokation, die nach jeder Definition und vor jeder Benutzung Spillcode einfügt. Sie erzeugt somit verhältnismäßig viel Spillcode, im Gegensatz zu den anderen im WCC vorhandenen Verfahren. Es kann nicht ausgeschlossen werden, dass eine solche naive Vorallokation auch die Modellierung des WCEP im ILP aus den vorherigen Kapiteln benachteiligt.

Insbesondere die Kostenschätzverfahren aus Abschnitt 5.2.4 gehen heuristisch bei der Bestimmung der WCET_{est} der Basisblöcke vor. Diese sind daher mit einem gewissen Fehler verbunden, welcher tendenziell größer wird, je mehr Spillcode eingefügt wird. Demnach kann der hohe Spillcodeanteil, den der Allokator RASA erzeugt, dazu führen, dass die fehlerhaften Basisblockkosten zu einer ILP-Modellierung führen, die eine Optimierungen in die falsche Richtung verursacht. Deshalb sollen andere Verfahren für die Vorallokation eingesetzt werden, die weniger Spillcode erzeugen, und sich somit positiv auf die Kostenschätzung auswirken.

Im Hinblick auf die Laufzeit schlägt Schmoll vor, ein Linear-Scan-Verfahren zu verwenden, das deutlich weniger Spillcode erzeugt und schnell arbeitet. Ein solcher Allokator existiert nicht im WCC , und er wird auch nicht im Rahmen dieser Diplomarbeit implementiert. Um die Auswirkung einer geänderten Vorallokation bewerten zu können, werden die anderen im WCC vorhandenen Registerallokatoren für eine Erstellung der Vorallokation benutzt.

- Der graphfärbungsbasierte RAGC
- Der codegrößenoptimierende RAILP-CS
- Der WCET-optimierende und graphfärbungsbasierte RAGC-WCET

Die beiden zuletzt genannten Registerallokatoren eignen sich aufgrund ihrer hohen Laufzeit eigentlich nicht für die Vorallokation, sollen aber trotzdem auf ihre Auswirkung hin untersucht werden.

7.3 Integration der Copy-Elimination

In Abschnitt 2.6.3 wird der Begriff der Copy-Elimination eingeführt und es wird erwähnt, dass Goodwin und Wilken in [GW96] eine Möglichkeit zur Implementierung vorschlagen. Sie sieht vor, dass für jeden Kopierbefehl, der ein virtuelles Register s_1 in ein anderes virtuelles Register s_2 kopiert, eine weitere Entscheidungsvariable $x^{elim}_{\{s_1,s_2\}\to p}$ hinzugefügt wird. Diese nimmt den Wert Eins an, wenn beide virtuellen Register s_1 und s_2 dem selben physikalischen Register p zugeordnet werden und der zugehörige Kopierbefehl entfernt werden kann. Deshalb werden der Variable $x^{elim}_{\{s_1,s_2\}\to p}$ negative Kosten zugewiesen, um den Gewinn durch das Löschen des Kopierbefehls zu berücksichtigen.

Die Formulierung von Goodwin und Wilken ist jedoch nur mit Hilfe umständlicher Veränderungen in das Registerallokationsverfahren von Schmoll zu integrieren. Alle Entscheidungsvariablen des Typs $x^{def}_{s\to p}$, $x^{cont}_{s\to p}$ und $x^{end}_{s\to p}$, die einen Kopierbefehl betreffen, müssten mit negativen Kosten in der Zielfunktion (Formel 5.10) berücksichtigt werden. Aufgrund dieser Tatsache wird eine Art der Implementierung gewählt, die einfacher zu realisieren und nicht so stark mit der ursprünglichen Modellierung verknüpft ist. Ein weiterer Vorteil ergibt sich aus der Wahlfreiheit, dieses Verfahren beliebig zum Allokationsvorgang hinzu- oder wegnehmen zu können.

Im Rahmen dieser Arbeit soll daher eine eigene Formulierung verwendet werden. Die grundsätzliche Idee einer neuen Entscheidungsvariable $x^{elim}_{\{s_1,s_2\}\to p}$ wird dazu beibehalten. Diese Variable soll im Folgenden Eliminationsvariable heißen. In einem ersten Schritt werden alle Kopierbefehle gesucht, die den Inhalt eines virtuellen Registers in ein anderes virtuelles Register kopieren. Demnach existiert für jeden solchen Befehl k ein Register, das benutzt wird und im Weiteren s^k_{use} heißen soll. Das andere Register s^k_{def} stellt das Zielregister dar, welches definiert wird. Im nächsten Schritt wird geprüft, ob die Entscheidungsvariablen $x^{def}_{s^k_{def}\to p}$ und $x^{cont}_{s^k_{use}\to p}$ zu einem

physikalischen Register p existieren. In diesem Fall wird eine Konjunktion beider Entscheidungsvariablen mit Hilfe der Transformation aus Formel 6.5 gebildet.

$$x^{elim}_{\{s^k_{def},s^k_{use}\}\to p} = x^{def}_{s^k_{def}\to p} \wedge x^{cont}_{s^k_{use}\to p} \tag{7.1}$$

Die Eliminationsvariable nimmt den Wert Eins an, wenn beide anderen Variablen den Wert Eins annehmen und damit beide virtuellen Register dem selben physikalischen Register p zugeordnet werden. Um solche Zuordnungen zu begünstigen, müssen alle Eliminationsvariablen in der Zielfunktion mit negativen Kosten berücksichtigt werden, was den Laufzeitgewinn durch die Entfernung des Kopierbefehls in die Lösung des ILPs miteinbezieht.

In der Formel 7.2 soll die Variable m_Z die neue Zielfunktion repräsentieren, die minimiert werden soll. Die Variable m_F entspricht der ursprünglichen Zielfunktion aus Formel 5.10, die den WCEP modelliert. Zusätzlich werden die Eliminationsvariablen für alle Kopierbefehle K und die entsprechenden physikalischen Register P aufsummiert und von den Kosten abgezogen. An dieser Stelle wird nicht wie in der Pipelineanalyse beachtet, ob ein Kopierbefehl parallel zu einem anderen Befehl ausgeführt werden kann, da der Kopierbefehl auf jeden Fall entfernt werden soll, egal, ob er Kosten erzeugt oder nicht.

$$m_Z = m_F - \sum_{k\in K}\sum_{p\in P} x^{elim}_{\{s^k_{def},s^k_{use}\}\to p} \tag{7.2}$$

Auf ähnlich einfache Weise kann die Copy-Elimination auch in den Registerallokator RAILP-CS integriert werden, da dieser die selbe ILP-Formulierung aus Abschnitt 5.1 als Grundlage verwendet. Dies wird jedoch in dieser Diploarbeit nicht beschrieben, da dieser Allokator nicht detailliert vorgestellt wird. Die Auswirkung der Integration werden aber in Kapitel 8 untersucht.

7.4 Kombinierte Zielfunktion

Um die Güte der Registerallokation bzgl. der WCET zu steigern, kann auch eine Kombination mit anderen Optimierungszielen wie der Codegröße sinnvoll sein. Zwar modelliert das ILP den WCEP und berücksichtigt auch dessen Instabilität, aber im Laufe der Modellierung werden einige Entscheidungen zur Vereinfachung der Kostenschätzung und Kostenmodellierung getroffen. Insbesondere überapproximiert die WCET-Analyse die WCET_{est} der einzelnen Basisblöcke, wodurch diese über der eigentlichen WCET liegt. Des Weiteren werden die unterschiedlichen Ausführungskontexte nicht berücksichtigt, stattdessen wird immer der Worst-Case für alle Ausführungen angenommen. Außerdem kann das modellierte ILP mehrere Lösungen besitzen, die die Zielfunktion optimieren. Diese können aber in ihrer Güte bzgl. der resultierenden WCET der fertigen Allokation voneinander abweichen.

Auf Grund dessen soll die Zielfunktion des ILPs mit codegrößenoptimierenden Elementen kombiniert werden. Das vorrangige Ziel dieser Maßnahme ist die Minimierung der resultierenden WCET. Zusätzlich verhindert eine solche Kombination aber auch das überflüssige Einfügen von Spillinstruktionen auf Nebenpfaden, die teilweise entstehen können, aber die WCET nicht beeinflussen. Insbesondere soll die Kombination den Lösungsraum des Optimierungsproblems derart einschränken, dass die Lösung in besseren Ergebnissen resultiert. Die codegrößenoptimierenden Elemente der Kombination entsprechen denen, die auch innerhalb der Zielfunktion des codegrößenoptimierenden Registerallokators (Abschnitt 3.2.2) verwendet werden.

Kombination 1: Eine solche Kombination lässt sich durch eine Erweiterung der Zielfunktion m_F aus Formel 5.10 modellieren. Benötigt wird hierzu eine ganzzahlige Variable m_C, die die Codegröße der fertigen Allokation repräsentiert. Diese wiederum soll wie die WCET minimiert werden, die durch m_F modelliert wird. Die Arbeitsweise des Registerallokators lässt bei der Modellierung der Codegröße einige Vereinfachungen zu, die die Lösung des ILPs nicht beeinflussen. Die vorhandenen Befehle des zu allokierenden Programmes werden mit Ausnahme der Kopierbefehle (Abschnitt 7.3) nicht entfernt, und es werden ausschließlich Spillinstruktionen hinzugefügt. Die Einbeziehung der Copy-Elimination in die Zielfunktion erfolgt am Ende des Abschnittes. Da nur 32-Bit-Speicher- und Ladebefehle einfügt werden, genügt es, deren Anzahl zu addieren, um eine Optimierung der Codegröße zu modellieren. Eine solche Summation kann durch das einfache Aufaddieren aller erstellten Entscheidungsvariablen $x^{st}_{s\to p}$ und $x^{ld}_{s\to p}$ realisiert werden, die angeben, ob die zugehörige Spillinstruktion eingefügt wird.

Die nachfolgende Nebenbedingung gibt die Formulierung einer kombinierten Zielfunktion wieder, wobei m_Z die neue Zielfunktion darstellt, die zu minimieren ist. Die beiden Faktoren a und b geben die Gewichtung der beiden Bestandteile an, welche beliebig variiert werden kann. In den nachfolgenden Messungen in Kapitel 8 soll die ursprüngliche Modellierung des WCEP jedoch einen höheren prozentualen Anteil an der Gesamtzielfunktion haben, um weiterhin die WCET als primäres Optimierungsziel beizubehalten.

$$m_Z = a \cdot m_F + b \cdot m_C \tag{7.3}$$

Kombination 2: Aus der Art der Implementierung der dynamischen Pipelineanalyse aus Abschnitt 5.2 eröffnet sich eine weitere Möglichkeit einer kombinierten Zielfunktion. Die Codegröße m_C wird wie zuvor beschrieben durch eine Summation aller Entscheidungsvariablen $x^{st}_{s\to p}$ und $x^{ld}_{s\to p}$ modelliert. Zusätzlich werden nun auch die Kosten für alle Spillentscheidungen summiert, wie dies Formel 7.4 zeigt. Diese Summe m_S kann zusammen mit der Codegröße m_C wie in Formel 7.5 kombiniert werden. Die beiden Faktoren a und b geben wiederum die Gewichtung der beiden Variablen an. Die grundsätzliche Modellierung des ILPs (Abschnitt 5.1) wird dabei beibehalten. Dieser Ansatz ist jedoch heuristisch und beachtet weder den WCEP

noch dessen Instabilität, und genauso wenig bezieht er die Ausführungsanzahl der einzelnen Spillinstruktionen mit ein. Wie alle Heuristiken kann dieser Ansatz auch zu deutlichen Verschlechterungen führen. Der große Vorteil dieser Methode liegt darin, dass weder eine Vorallokation noch eine WCET-Analyse ebendieser notwendig ist. Dadurch ist dieses Verfahren im Gegensatz zum erstgenannten unabhängig von der Güte einer statischen WCET-Analyse.

$$m_S = \sum_{E \text{ ist Spillentscheidung}} (c_k^E + c_s^E) - \sum_{sb \text{ ist Spillblock}} c_e^{sb} \tag{7.4}$$

$$m_Z = a \cdot m_S + b \cdot m_C \tag{7.5}$$

Integration der Copy-Elimination: Die Copy-Elimination aus Abschnitt 7.3 kann für beide Kombinationen zusätzlich ausgeführt werden. Berücksichtigt man deren Modellierung in Formel 7.2, so ergeben sich aus den Formeln 7.3 und 7.5 folgende neue Zielfunktionen.

$$m_Z = a \cdot m_F + b \cdot m_C - \sum_{k \in K} \sum_{p \in P} x^{elim}_{\{s^k_{def}, s^k_{use}\} \to p} \tag{7.6}$$

$$m_Z = a \cdot m_S + b \cdot m_C - \sum_{k \in K} \sum_{p \in P} x^{elim}_{\{s^k_{def}, s^k_{use}\} \to p} \tag{7.7}$$

„Ingenieure müssen den Code in Bezug auf die Worst-Case Execution Time (WCET) analysieren.“

John L. Hennessy und David A. Patterson
übersetzt aus dem Englischen [HP07, Seite D-4]

8 Auswertung

Nachdem alle Implementierungen vorgestellt sind, die im Rahmen dieser Diplomarbeit umgesetzt werden, sollen nun die Ergebnisse des Benchmarkings präsentiert werden. Schrittweise sollen hierzu die Erweiterungen zu dem bestehenden Verfahren aus Kapitel 5 hinzugenommen und ausgewertet werden. Zunächst sollen die Erweiterungen aus Kapitel 7 der Reihe nach hinzugefügt werden, wie sie auch innerhalb des Kapitels vorkommen. Dies geschieht im Abschnitt 8.2. Ausgenommen sei hier die Kombination der Zielfunktion aus Abschnitt 7.4, diese wird erst später im Abschnitt 8.4 untersucht. Nach den Erweiterungen werden die Auswirkungen der beiden Pipelineanalyseverfahren aus Kapitel 6 in Abschnitt 8.3 betrachtet.

Währenddessen werden bzgl. der resultierenden WCET immer wieder Vergleiche zu den bestehenden Registerallokationsverfahren gezogen, insbesondere auch den hier behandelten Registerallokator in seiner Grundversion zu Beginn dieser Arbeit. Des Weiteren werden die Laufzeiten der Verfahren in Abschnitt 8.5 gegenüber gestellt. Obendrein finden die Auswirkungen auf die ACET bzw. der Größe des ILPs im Abschnitt 8.6 bzw. 8.7 Berücksichtigung. Zunächst soll im ersten Abschnitt dieses Kapitels erläutert werden, wie die aufgenommenen Messwerte erzeugt werden. Die verwendeten Verfahren und Benchmarks, soweit noch nicht vorgestellt, finden hier Erwähnung.

8.1 Messverfahren

Will man den Effekt einer Optimierung innerhalb eines Compilers bestimmen, so führt man die Übersetzung einmal mit und ohne sie aus. Aus dem Unterschied bzgl. des Gütekriteriums lässt sich die Güte der Optimierung bestimmen. Im Gegensatz zu anderen Optimierungen muss die Registerallokation immer ausgeführt werden, sofern virtuelle Register verwendet werden. Sie kann daher nicht beliebig weg- oder hinzugeschaltet werden. Deshalb werden Registerallokationsverfahren immer einander gegenüber gestellt, um Aussagen über ihre Güte treffen zu können.

Im Folgenden wird daher das in dieser Arbeit behandelte Registerallokationsverfahren (RAILP-WCET) den anderen innerhalb des WCC verfügbaren und in Abschnitt 3.2.2 erwähnten Verfahren gegenübergestellt. Die im WCC enthaltenen

Registerallokatoren sollen der Übersichtlichkeit halber noch einmal kurz zusammen mit ihren Kürzeln genannt werden.

- graphfärbungsbasierter Registerallokator (RAGC)
- codegrößenoptimierender Registerallokator (RAILP-CS)
- graphfärbungsbasierter WCET-optimierender Registerallokator (RAGC-WCET)
- naiver Registerallokator (RASA)

Letzterer wird, wie schon erwähnt, nur zur Vorallokation genutzt. Die anderen drei Verfahren sind eigenständig und werden zu Vergleichen herangezogen. Insbesondere verwendet der RAILP-CS das selbe Prinzip der ILP-Modellierung (Abschnitt 5.1) wie der RAILP-WCET, weshalb sich die Zielfunktionen beider Ansätze leicht kombinieren lassen.

Die im WCC der Registerallokation vorhergehenden Verfahren verhalten sich deterministisch, daher wird jeder der verschiedenen Allokatoren auf dieselbe Zwischendarstellung des jeweiligen Programmcodes angewendet. Des Weiteren erfolgen alle nachstehenden Messungen unter Verwendung der Optimierungsstufe O3. Zur Messung werden handelsübliche PC-Systeme mit Linux-Betriebssystemen verwendet. Zusätzlich wird folgende Software benutzt.

- ILP-Solver CPLEX (Abschnitt 2.3)
- WCET-Analyseprogramm a^3 Build: b126886 (Abschnitt 3.3)

Für die Erzeugung der Messwerte wird die Benchmark-Suite WCETBENCH des WCC verwendet, welche sich aus weit über 100 Einzelbenchmarks zusammensetzt. Diese Benchmarks entstammen verschiedenen anderen Benchmark-Suiten. Sie decken verschiedene Anwendungsgebiete innerhalb Eingebetteter Systeme und den ganzen Variationsbereich dort verwendeter Software ab. In Anhang B sind jene Benchmarks aufgelistet, die im Rahmen der Messung für diese Diplomarbeit benutzt werden. Kurzbeschreibungen der aufgenommenen Messreihen sind in Anhang C zu finden. Die Ergebnisse dieser Messreihen sind in den nachfolgenden Anhängen D bis L tabellarisch dargestellt.

Die Lösung eines ILPs mit CPLEX kann je nach Fall ziemlich viel Rechenzeit in Anspruch nehmen. Für die Aufnahme der Messwerte wird deswegen eine feste Zeitschranke von 40 Minuten benutzt, die maximal zur Lösung des ILPs verwendet werden darf. Wird diese Zeitschranke überschritten wird der Prozess abgebrochen. Für einige Benchmarks werden mehrere ILPs erzeugt, da sie über mehrere Funktionen verfügen und jede einzeln betrachtet wird. Die Zeitmessung beginnt mit jeder Lösung eines einzelnen ILPs neu. Nun kann es sein, dass der ILP-Solver schon eine gültige Lösung gefunden hat, die jedoch nicht optimal sein muss. In diesem Fall ist der entsprechende Messwert unterstrichen, der sich in einer der Tabellen des Anhangs befindet. Liegt jedoch zum Zeitpunkt des Abbruchs keine gültige Lösung vor, die alle Nebenbedingungen erfüllt, wird auch der komplette Übersetzungsvorgang

abgebrochen. Ein solcher Fall wird jedoch während der Messung nicht beobachtet, da die Zeitschranke hoch genug gewählt ist, dass alle ILPs eine gültige Lösung erhalten.

Wie schon bei der Vorallokation wird auch die WCET nach der endgültigen Registerallokation durch das Analyseprogramm a^3 bestimmt, das in Abschnitt 3.3 vorgestellt wird. Noch zu erwähnen ist die Bestimmung der ACET, die innerhalb des WCC mit Hilfe des Entwicklungswerkzeugs CoMET[5] der Firma VaST Systems Technology geschieht. Dieses stellt virtuelle Hardwaremodelle für komplexe Eingebettete Systeme zur Verfügung, auf denen Simulationen durchgeführt werden können. Das Verfahren soll hier jedoch nicht im Detail dargestellt werden.

8.2 Auswirkungen der Erweiterungen

Das Hauptaugenmerk liegt bei allen folgenden Messungen auf dem veränderten Verhalten des RAILP-WCET, das auf den einzelnen Erweiterungen aus den Kapiteln 6 und 7 beruht. Im Folgenden wird zunächst ausschließlich die einfache Kostenschätzung untersucht, die aufwändige Kostenschätzungen findet in diesem Kapitel keinerlei Beachtung, da kein signifikanter Unterschied bzgl. der Güte der resultierenden WCET zwischen beiden Verfahren auszumachen ist. Alle Durchschnittswerte, die im Folgenden auftreten, werden über allen Werten der zugehörigen Messreihe gebildet und entsprechen nicht dem Durchschnitt der im Diagramm dargestellten Benchmarks. Des Weiteren entspricht der RAGC der relativen Bezugsgröße innerhalb aller Diagramme.

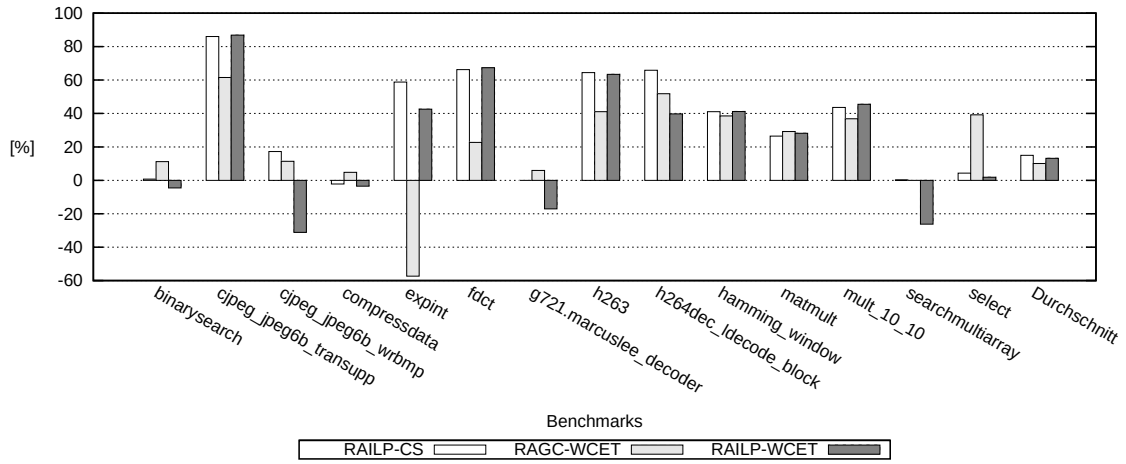


Abbildung 8.1: Verhalten der Registerallokatoren zu Beginn der Diplomarbeit

Das Diagramm 8.1 zeigt beispielhaft für einige Benchmarks die Verbesserung bzgl. der WCET, die aus der Anwendung der verschiedenen Registerallokationsverfahren

[5] http://www.synopsys.com/Tools/SLD/VirtualPrototyping/Pages/CoMET-METeor.aspx

des WCCs folgt. Die Funktionalität der einzelnen Verfahren entspricht dem Zustand zu Beginn dieser Diplomarbeit. Die Allokatoren RAGC und RAGC-WCET werden im Rahmen dieser Diplomarbeit nicht verändert. Wohingegen in den RAILP-CS im Laufe der Messungen die Copy-Elimination aus Abschnitt 7.3 integriert wird. Die Werte des RAILP-WCET werden durch das Registerallokationsverfahren erzeugt, das in Kapitel 5 beschrieben ist und welches der abschließenden Implementierung von Schmoll entspricht. Hierzu wird die einfache Kostenschätzung aus Abschnitt 5.2.4 verwendet. Die dargestellten Werte beruhen auf den Messwerten aus Tabelle D.1, und sie geben die Verbesserung bzgl. der WCET gegenüber RAGC an. Man kann erkennen, dass RAILP-WCET für die meisten Benchmarks bessere Ergebnisse erzeugt als RAGC, aber auch einige Ausnahmen wie z. B. *cjpeg_jpeg6b_wrbmp* existieren, bei denen es zu markanten Verschlechterungen kommt. Im Durchschnitt kann RAILP-CS mit einem Wert von 14,9 % die höchste Verbesserung erzielen. Die von RAILP-WCET erreichte durchschnittliche Verbesserung liegt knapp darunter bei 13,2%, obwohl dieser Ansatz im Gegensatz zu RAILP-CS die WCET gezielt zu minimieren versucht. Ziel dieser Diplomarbeit ist es daher, die Güte des RAILP-WCET zu steigern und die WCET noch stärker zu minimieren. RAGC-WCET kommt als heuristischer Ansatz sogar auf einen Durchschnittswert von 10,0 %, und liegt damit aber trotzdem deutlich hinter den beiden anderen Verfahren.

Die folgenden Messungen bauen schrittweise aufeinander auf. Einmal vorgenommene Erweiterungen oder Entscheidungen werden daher in den Folgemessungen fortgeführt.

8.2.1 Auswirkungen der Verwendung des Scratchpad-Speichers

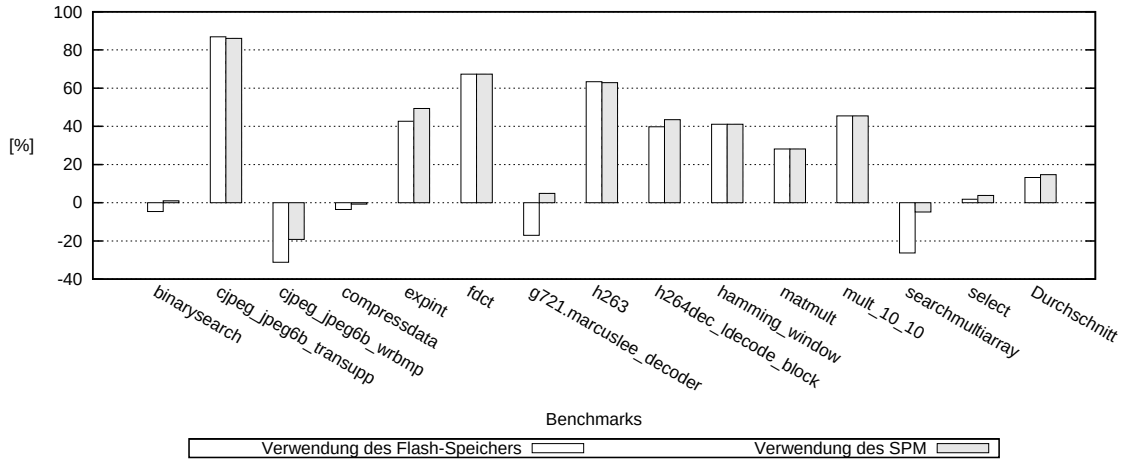


Abbildung 8.2: Auswirkungen der Verwendung des Scratchpad-Speichers

Das Diagramm 8.2 belegt welche Verbesserungen für RAILP-WCET auftreten, wenn anstelle des Flash-Speichers (Tabelle D.1 vierte Spalte) der SPM (Tabelle E.1

erste Spalte) wie Abschnitt 7.1 beschrieben, zur Analyse genutzt wird. Man kann hier deutlich den positiven Effekt auf die erzielten Ergebnisse erkennen. Ausnahmen bilden hier nur die Benchmarks *cjpeg_jpeg6b_transupp* und *h263*, für die leichte Einbußen entstehen. Trotzdem wird eine durchschnittliche Verbesserung von 1,5 % bzw. 14,7 % zur Verwendung des Flash-Speichers bzw. zum RAGC erlangt. Deshalb wird der SPM auch im Weiteren durchgängig verwendet.

8.2.2 Auswirkungen einer veränderten Vorallokation

Der Abschnitt 7.2 sieht als nächstes einen Austausch der Vorallokation vor. Die Auswirkungen eines solchen Austauschs veranschaulicht Diagramm 8.3. Bei Verwendung der verschiedenen Vorallokationen RASA, RAGC, RAILP-CS bzw. RAGC-WCET werden durchschnittliche Verbesserungen von 14,7 %, 15,1 %, 16,0 % bzw. 15,5 % erzielt (Tabelle E.1). Hier wird in besonderem Maße deutlich, dass sich die Güte der Vorallokation auf die Güte der resultierenden WCET auswirkt. An dieser Stelle muss beachtet werden, dass die positiven Effekte durch die Verwendung des SPM hier zusätzlich schon enthalten sind. Die Werte für RASA entsprechen denen, die in Diagramm 8.2 zuvor angeführt werden. Für die drei anderen Vorallokationen ergeben sich weitere Verbesserungen. Vor allem RAILP-CS zeichnet sich durch eine starke Verbesserung aus. Nichtsdestotrotz wird von hier an RAGC als Vorallokation benutzt, weil er weniger Laufzeit benötigt und bessere Ergebnisse liefert als RASA. Die anderen beiden Verfahren werden aufgrund ihrer hohen Laufzeit nicht verwendet, obwohl sie noch bessere Ergebnisse generieren.

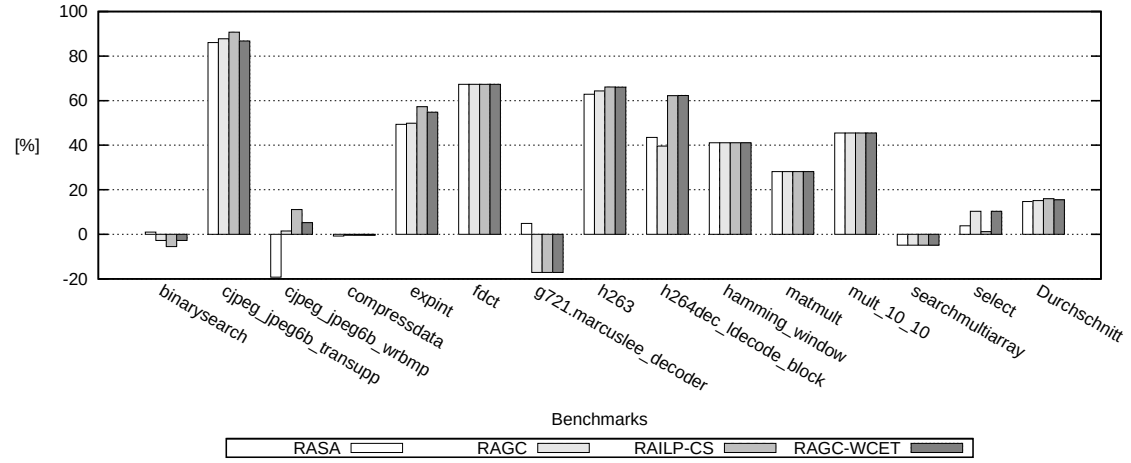


Abbildung 8.3: Auswirkungen einer veränderten Vorallokation

Betrachtet man die Laufzeiten aus Tabelle E.2, so ergibt sich mit dem RAGC als Vorallokation eine Laufzeit von durchschnittlich ungefähr 2 Minuten und 27 Sekunden. Diese Zeit umfasst die Ausführungsdauer der Vorallokation, deren Auswertung, die Erstellung des ILPs, dessen Lösung sowie die Fertigstellung der endgültigen

Allokation. Die anderen drei Verfahren benötigen mehr Rechenzeit (RASA 3:26, RAILP-CS 4:21, RAGC-WCET 6:03). Überraschenderweise führt die Verwendung des naiven Registerallokators RASA tatsächlich zu einer höheren Ausführungszeit als die des RAGC , was auf die im Vergleich höhere Ausführungsdauer der statischen WCET-Analyse zurückzuführen ist.

8.2.3 Auswirkungen der Copy-Elimination

Durch die Integration der Copy-Elimination aus Abschnitt 7.3 werden weitere Verbesserungen erzielt. Das Diagramm 8.4 stellt diese für die beiden Registerallokatoren RAILP-CS und RAILP-WCET dar (Tabelle F.1). Nahezu jeder Benchmark erfährt eine Verbesserung gegenüber der Verwendung des RAGC. Die einzige Ausnahme der aufgeführten Benchmarks bildet *searchmultiarray*. Für RAILP-CS werden nun durchschnittlich Verbesserungen von 19,0 % erreicht. RAILP-WCET ist durchschnittlich 16,3 % besser als RAGC und liegt damit bzgl. der Güte über den ursprünglichen Ansätzen des RAILP-CS und RAILP-WCET aus Diagramm 8.1. Er liegt jedoch auch deutlich unter dem Wert, den RAILP-CS mit Copy-Elimination erzeugt. Eine nennenswerte Auswirkung auf die Laufzeit ist durch die Integration nicht entstanden (Tabelle F.1). Für RAILP-CS kann die Laufzeit im Durchschnitt sogar leicht gesenkt werden.

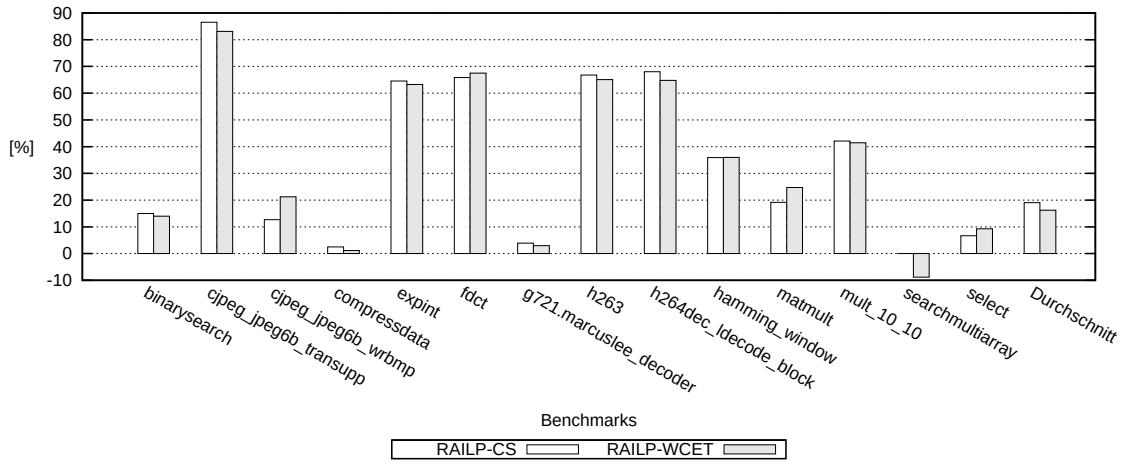


Abbildung 8.4: Auswirkungen der Copy-Elimination

Insgesamt können ohne integrierte Copy-Elimination 10,3 % bzw. 20,8 % der Kopierbefehle im Anschluss an die Registerallokation entfernt werden, wenn RAILP-CS bzw. RAILP-WCET benutzt wird (Tabelle G.1). Mit der Hinzunahme der Copy-Elimination können diese Werte auf 46,7 % und 53,0 % gesteigert werden.

8.3 Auswirkungen der Pipelineanalysen

Diagramm 8.5 zeigt die Verbesserungen für RAILP-WCET gegenüber RAGC, wenn die statische und dynamische Pipelineanalyse aus Kapitel 6 verwendet wird. Für die WCET-Werte aus Tabelle H.1 folgt eine durchschnittliche Verbesserung von 16,2 %. Dieser Wert liegt knapp unter dem aus Diagramm 8.4. Allerdings liegt er auch deutlich über dem anfänglichen Wert des RAILP-WCET aus Diagramm 8.1.

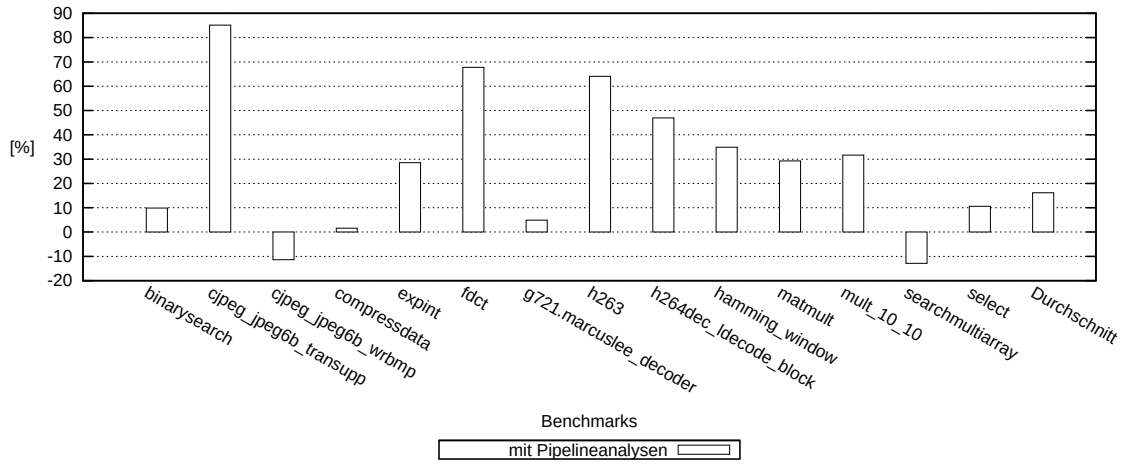


Abbildung 8.5: Auswirkungen der Pipelineanalysen

Wieder einmal offenbaren sich Schwächen in Bezug auf die beiden Benchmarks *cjpeg_jpeg_wrbmp* und *searchmultiarray*. Diese scheinen sehr anfällig gegen Verschlechterungen zu sein. Mögliche Gründe für diese Verschlechterungen sollen nicht in diesem Kapitel diskutiert werden, sie finden daher in der Bewertung im Abschnitt 9.2 Platz. Ungeachtet der leichten Verschlechterungen für den RAILP-WCET sollen die Pipelineanalysen auch in den nachfolgenden kombinierten Ansätzen verwendet werden, da sich insbesondere herausgestellt hat, dass diese Verfahren einen positiven Effekt auf den ersten kombinierten Ansatz haben. Der zweite kombinierte Ansatz ist ohnehin nur zusammen mit der dynamischen Pipelineanalyse umsetzbar.

8.4 Auswirkungen einer kombinierten Zielfunktion

In diesem Abschnitt sollen die Auswirkungen der beiden kombinierten Ansätze aus Abschnitt 7.4 untersucht werden. Diagramm 8.6 zeigt die Verbesserungen, die durch Anwendung der ersten Kombination (Formel 7.6) entstehen. Die Faktoren a und b werden im Verhältnis 70:30, 80:20, 90:10 und 99:1 variiert, um zu modellieren, dass die WCEP-Minimierung das Primärziel der kombinierten Zielfunktion sein soll. Der Faktor a stellt die Gewichtung der WCEP-Modellierung dar, während b die Gewichtung der Codegröße modelliert. Es ist zu erkennen, dass mit Ausnahmen

des *searchmultiarray* für jeden Benchmark im Diagramm Verbesserungen erreicht werden, die teilweise sogar die 60- oder 80-Prozent-Marke überschreiten.

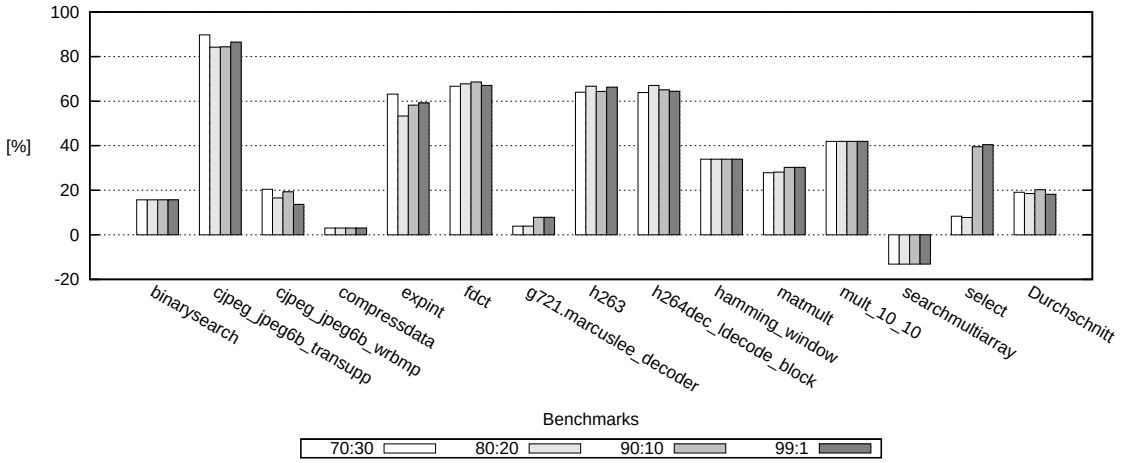


Abbildung 8.6: Auswirkungen der ersten Kombination

Aus der Betrachtung des Diagramms lässt sich kein Verhältnis für die beiden Faktoren ausmachen, das herausragende WCET-Werte erzeugt. Durch eine Auswertung der Tabelle I.1, die die Werte der zugehörigen Messungen enthält, stellt sich heraus, dass das Verhältnis 90:10 die besten Ergebnisse produziert. Hier liegt die durchschnittliche Verbesserung bei 20,3 %. Dies stellt den besten aller aufgenommenen Durchschnittswerte dar. Insbesondere liegt dieser Wert auch über dem des RAILP-CS im Diagramm 8.4. Zudem übertrifft er auch alle von der zweiten Kombination aus Diagramm 8.7 erzeugten Werte. Zu erwähnen bleiben noch die genauen Durchschnittswerte der übrigen Verhältnisse. So produziert das Verhältnis 70:30, 80:20 bzw. 99:1 eine Verbesserung von 19,1 %, 18,5 % bzw. 18,2 % gegenüber dem RAGC.

Die Auswirkungen der zweiten Kombination (Formel 7.7) präsentiert Diagramm 8.7. Für die Faktoren a und b werden die Verhältnisse 1:99, 20:80, 50:50, 80:20, 99:1 benutzt, da hier nicht im Vorfeld ein Primärziel festgelegt werden soll. Der Faktor a stellt die Gewichtung der erzeugten Spillkosten dar, b hingegen bestimmt die Gewichtung der Codegröße. Alle zugehörigen WCET-Werte der Messungen sind in Tabelle J.1 hinterlegt. Bis auf die minimale Verschlechterung beim Benchmark *compressdata* erzeugt dieses Verfahren für die aufgezeigten Benchmarks nur Verbesserungen, die teilweise sehr deutlich sind. Für *searchmultiarray* werden im Gegensatz zur ersten Kombination keine Verschlechterungen mehr generiert.

Bei Betrachtung des Diagramms lässt sich keine Aussage darüber treffen, welches Verhältnis vorzuziehen ist. Die Auswertung der Messwerte ergibt, dass das Verhältnis 80:20 mit einer durchschnittlichen Verbesserung von 19,6 % zum RAGC die besten Resultate liefert. Das Verhältnis 1:99, 20:80, 50:50 bzw. 99:1 führt zu durchschnittlichen Verbesserungen von 19,1 %, 18,7 %, 19,2 % bzw. 18,0 % gegenüber RAGC.

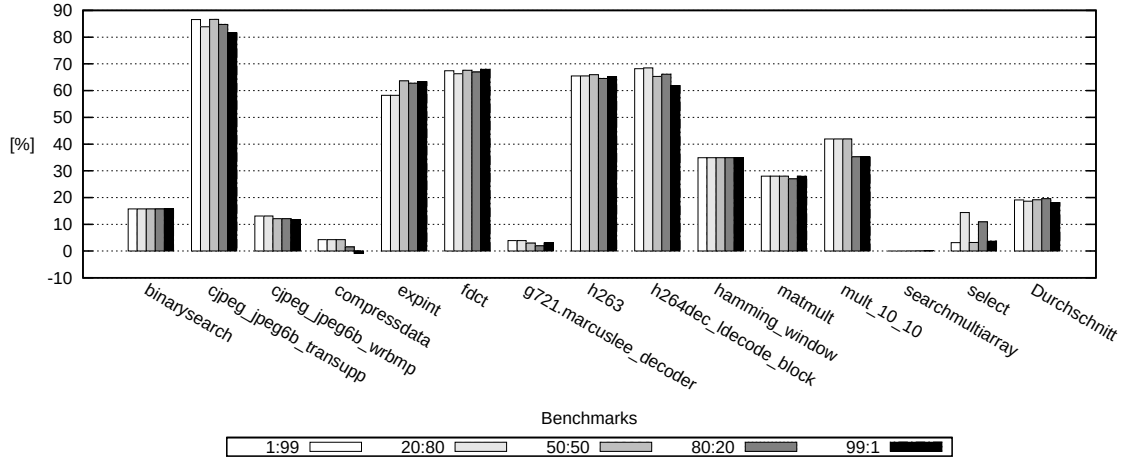


Abbildung 8.7: Auswirkungen der zweiten Kombination

In den nachfolgenden Diagrammen werden die erste und zweite Kombination weitergehend untersucht. Dabei werden jene Verhältnisse weiter betrachtet, die die jeweils besten Ergebnisse erbracht haben. Für die erste Kombination entspricht dies dem Verhältnis 90:10, und für die zweite Kombination wird das Verhältnis 80:20 weiter betrachtet. Bemerkenswert ist, dass beide Verhältnisse die Kosten der Spillinstruktionen deutlich stärker als die Codegröße gewichten.

8.5 Laufzeitvergleich

Bevor die Laufzeiten der einzelnen Verfahren miteinander verglichen werden können, muss an dieser Stelle klar gestellt sein, was in die Laufzeit einfließen soll. Für alle Registerallokationsverfahren soll gelten, dass die Ausführungszeit vorhergehender und nachfolgender Übersetzungs- und Optimierungsschritte nicht beachtet werden soll. Die Ausführungszeit des RAGC entspricht der Zeit, die zur Lösung des Graphfärbeproblems und zur Übertragung der Allokationsergebnisse benötigt wird. Beim RAGC-WCET kommen die Ausführungszeiten der zwischenzeitlichen WCET-Analysen durch a^3 sowie die mehrfache Erstellung und Lösung von ILPs hinzu. Auch beim RAILP-CS fließt die Zeit zur Erstellung und Lösung eines ILPs sowie der Übertragung der Ergebnisse mit in die Laufzeit ein.

Beim ebenfalls ILP-basierten RAILP-WCET kommt noch die Erstellung einer Vorallokation, deren WCET-Analyse sowie die Kostenschätzung hinzu. Ebenso verhält es sich mit der darauf aufbauenden ersten Kombination. Die zweite Kombination verhält sich insgesamt wie RAILP-CS, da keine Vorallokation benötigt wird.

Mit einer durchschnittlichen Ausführungszeit von 0,26 Sekunden stellt der RAGC das mit großem Abstand schnellste Verfahren dar (Tabelle D.2). Wie schon bei den vorhergehenden Diagrammen geben die folgenden Diagramme alle Ausführungszei-

ten relativ zur Graphfärbung an. Aufgrund der großen Laufzeitunterschiede werden die Werte auf einer logarithmischen Skala aufgetragen.

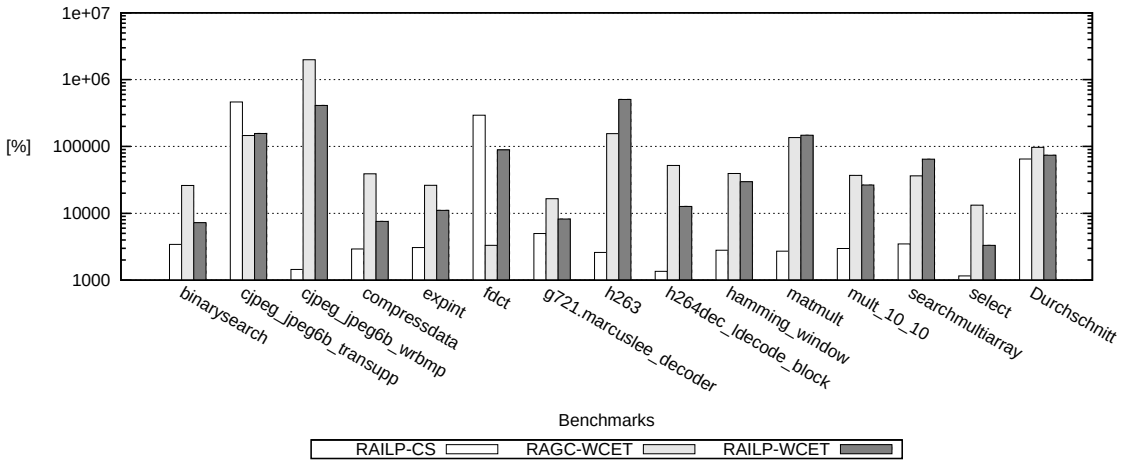


Abbildung 8.8: Vergleich der Ausführungszeiten zu Beginn der Arbeit

Diagramm 8.8 zeigt die Ausführungszeiten (Tabelle D.2) der in Diagramm 8.1 behandelten Verfahren. Nach RAGC ist RAILP-CS das mit durchschnittlich 2 Minuten und 49 Sekunden zweitschnellste Verfahren, gefolgt von RAILP-WCET mit 3 Minuten 13 Sekunden und RAGC-WCET mit 4 Minuten 13 Sekunden. Alle drei Verfahren benutzen während ihrer Ausführung ILPs, was sich in deutlich höheren Ausführungszeiten im Gegensatz zum RAGC ausdrückt. Besonders jene Benchmarks, die während der Lösung des ILPs die festgelegte Zeitschranke von 40 überschreiten, sorgen dafür, dass die Durchschnittswerte stark ansteigen. Solche Benchmarks sind, wie zuvor schon erwähnt, durch Unterstreichungen in den Tabellen des Anhangs markiert.

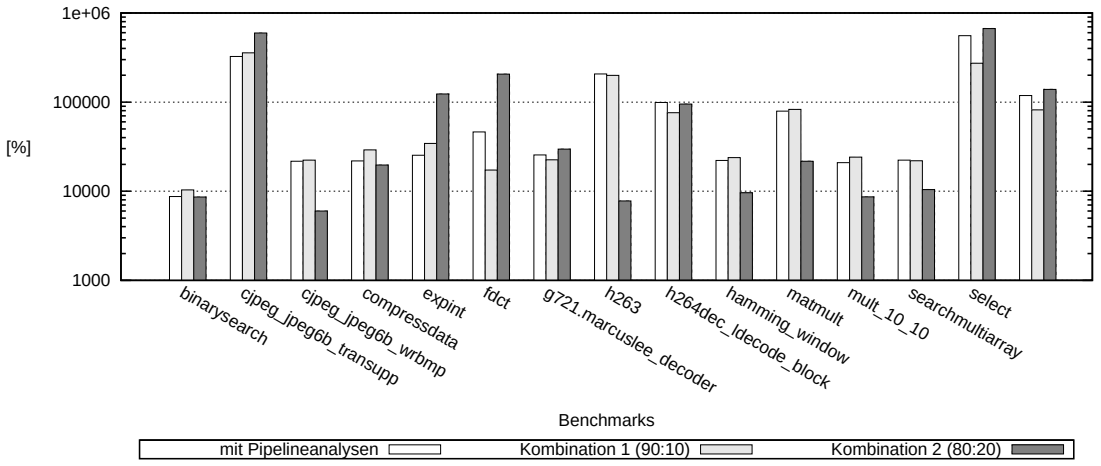


Abbildung 8.9: Vergleich der Ausführungszeiten des veränderten RAILP-WCET

Im Diagramm 8.9 sind die Laufzeiten des RAILP-WCET unter Verwendung der Pipelineanalysen sowie die Laufzeiten der beiden Kombinationen wiedergegeben. Aus der Betrachtung des Diagramms kann geschlossen werden, dass die erste Kombination mit einem Verhältnis von 90:10 für die Faktoren a und b mit 3 Minuten 33 Sekunden (Tabelle I.2 dritte Spalte) das durchschnittlich schnellste Verfahren darstellt. Danach folgt der unkombinierte Ansatz, der die Pipelineanalyseverfahren verwendet, mit 3 Minuten 29 Sekunden (Tabelle H.1 zweite Spalte). Die zweite Kombination mit einem Verhältnis von 80:20 stellt mit einer durchschnittlichen Laufzeit von 6 Minuten 1 Sekunde (Tabelle J.2 vierte Spalte) das deutlich langsamste Verfahren dar, obwohl es für etliche Benchmarks wie z. B. *cjpeg_jpeg6b_wrbmp* und *h263* auch deutlich bessere Werte erzeugt als die beiden anderen Verfahren. Wieder einmal wird die Bildung des Mittelwertes durch jene Benchmarks dominiert, die sehr lange Ausführungszeiten benötigen und teilweise auch die Zeitschranke zur Lösung des ILPs überschreiten. Solche Fälle treten bei der zweiten Kombination im Gegensatz zur ersten vermehrt auf.

8.6 ACET-Messung

Neben der resultierenden WCET der verschiedenen Verfahren soll auch die ACET Beachtung finden, da diese auch von großer Bedeutung ist. Die ACET steht in direktem Zusammenhang mit der Güte der Registerallokation und ganz besonders mit der Größe des Spillcode-Overheads. Eine gute Registerallokation kann die ACET deutlich reduzieren. So gelingt es dem RAILP-CS bzw. RAGC-WCET die durchschnittliche ACET im Vergleich zum RAGC, um 12,0 % bzw. 7,9 % zu senken, wie dies Diagramm 8.10 veranschaulicht. Die hier dargestellten Durchschnittswerte beruhen auf den Messwerten aus Anhang K.

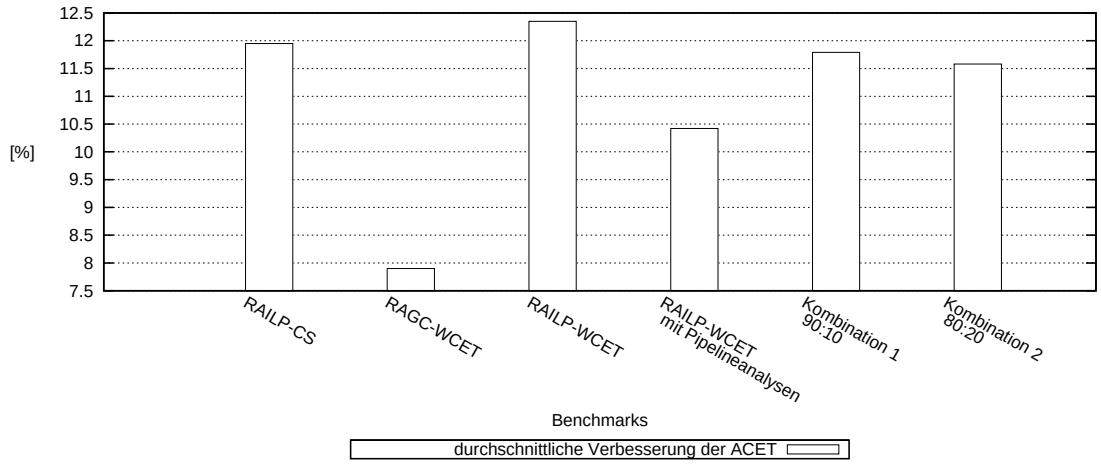


Abbildung 8.10: Verbesserungen bzgl. der ACET

Bei Betrachtung des Verfahrens das dem RAILP-WCET zugrunde liegt, lässt sich nur schwer eine Vorhersage treffen, wie sich das Verfahren wohl auf die resultierende ACET auswirkt. So könnte vermutet werden, dass diese höher liegt als beim RAILP-CS, da zur Minimierung des WCEP unter Umständen viel Spillcode auf nebenläufigen Pfaden eingefügt werden kann. In der Tat führt die Grundversion des RAILP-WCET, wie sie zu Beginn dieser Diplomarbeit vorliegt, jedoch zu einer noch stärkeren Reduktion der ACET. Im Durchschnitt wird eine Verbesserung von 12,4 % erzielt. Werden die pipelinebasierten Verfahren angewendet, ergibt sich ein Wert von 10,4 %. Die beiden Kombinationen führen wiederum zu besseren durchschnittlichen ACET-Werten. Für die erste Kombination ergibt sich eine Verbesserung von durchschnittlich 11,8 %. Der Wert für die zweite Kombination beträgt 11,6 %.

8.7 Größe des ILPs

Goodwin und Wilken bestimmen in ihrer Arbeit [GW96] die Anzahl der eingefügten Nebenbedingungen experimentell. Sie geben mit $\mathcal{O}(i^{1,3})$ eine Größenordnung für die Anzahl der eingefügten Nebenbedingungen an, wobei i der Anzahl der Instruktionen entspricht. In Diagramm 8.11 ist für RAILP-WCET die Anzahl der Nebenbedingungen gegen die Anzahl der vorhandenen Instruktionen innerhalb der Zwischendarstellung für alle Benchmarks aufgetragen. Das zugehörige Verfahren entspricht dem anfänglichen Ansatz aus Diagramm 8.1. In das Diagramm ist zusätzlich eine polynomielle Funktion $g(x) \in \mathcal{O}(i^{1,3})$ eingezeichnet. Da die Messpunkte sich um $g(x)$ gruppieren, können die Messergebnisse von Goodwin und Wilken bestätigt werden. Der Übersicht halber wird an dieser Stelle eine logarithmische Skalierung verwendet. Alle in diesem Abschnitt verwendeten Messwerte sind in Tabelle L.1 hinterlegt.

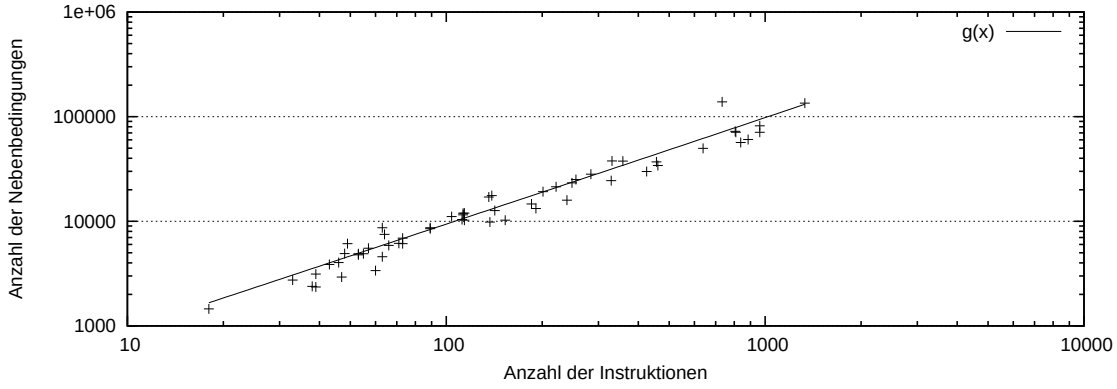


Abbildung 8.11: Größe des ILPs zu Beginn

Durch die Verwendung der dynamischen Pipelineanalyse innerhalb des ILPs steigt die Anzahl der erzeugten Nebenbedingungen, was in Diagramm 8.12 zu erkennen

ist. Wiederum wird eine polynomielle Funktion $g(x) \in \mathcal{O}(i^{1,3})$ eingetragen, wobei diesmal der lineare Term zusätzlich mit einem Faktor 2,4 multipliziert wird. Aus der Tatsache, dass sich die Messpunkte um die Funktion gruppieren, kann gefolgert werden, dass es sich hier um einen linearen Zuwachs handelt. Dies erscheint durchaus nachvollziehbar, wenn man bedenkt, dass die zu einer Instruktion erzeugten Nebenbedingungen nur sehr lokal das Umfeld der Instruktion modellieren, und niemals Eigenschaften von Instruktionen berücksichtigen, die weiter entfernt im Programmcode stehen.

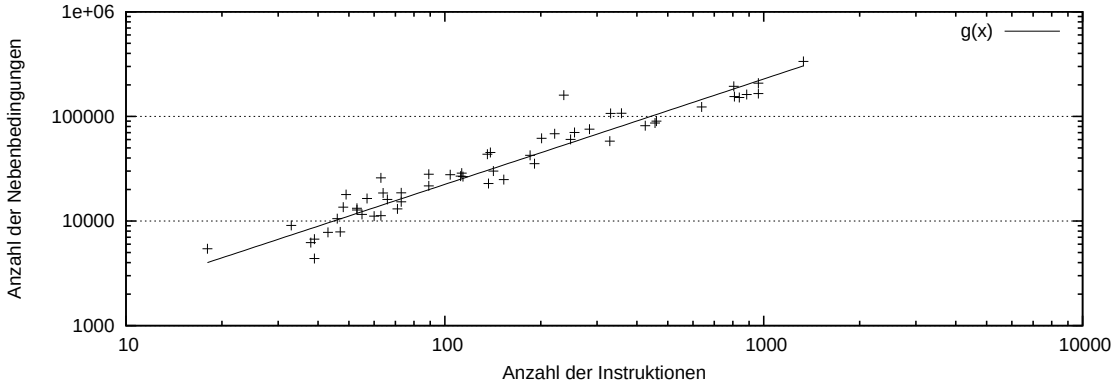


Abbildung 8.12: Größe des ILPs mit dynamischer Pipelineanalyse

Eine separate Darstellung für die erste Kombination ist nicht notwendig, weil dieses Verfahren genauso viele Nebenbedingungen erzeugt, wie das in Diagramm 8.12 dargestellte Verfahren. In Diagramm 8.13 ist für die zweite Kombination die Anzahl der Nebenbedingungen gegen die Anzahl der Instruktionen aufgetragen. Auch hier lässt sich ein linearer Zusammenhang bzgl. der Steigerung der Anzahl der Nebenbedingungen des entstehenden ILPs beobachten. Wiederum gruppieren sich die Messwerte um eine polynomielle Funktion $g(x) \in \mathcal{O}(i^{1,3})$, wobei der lineare Term im Vergleich zu Diagramm 8.11 mit einem Faktor 2,7 multipliziert wird.

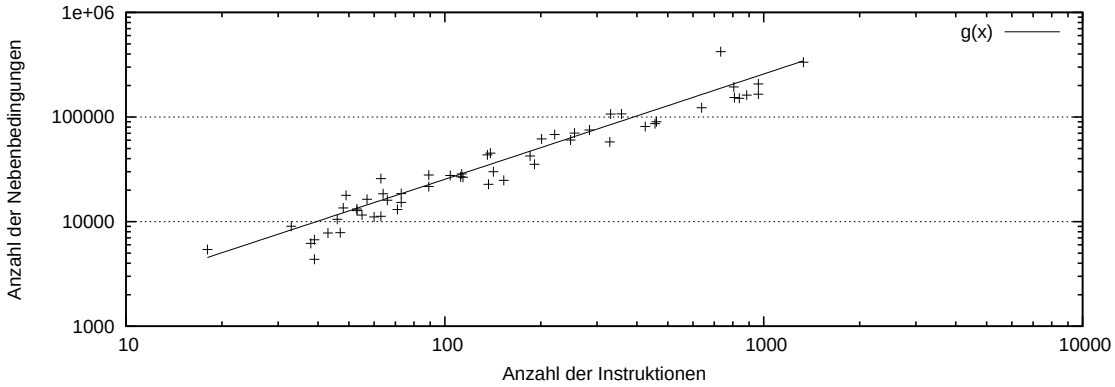


Abbildung 8.13: Größe des ILPs aus Kombination 2

„Die vermehrte Benutzung fortgeschrittener Prozessor-Hardware hat den Bedarf nach aufwendigeren Hardware-Modellen geweckt. Um z. B. das Zeitverhalten von Caches oder Pipelines auszudrücken und zu modellieren, werden Erweiterungen anderer Ansätze benötigt.“

Raimund Kirner und Peter Puschner
übersetzt aus dem Englischen [KP05, Seite 190]

9 Fazit

In dieser Diplomarbeit werden verschiedene ILP-basierte Registerallokationsverfahren vorgestellt, die das Ziel haben, die WCET der zu übersetzenden Programme zu minimieren. Als Grundlage dient die Registerallokation aus [Sch08], auf der alle hier präsentierten Erweiterungen aufbauen. Das ursprüngliche Verfahren wies Schwächen bzgl. der Güte der resultierenden WCET-Werte auf. Diese sollen im Rahmen dieser Arbeit beseitigt werden, was sich möglichst in einer niedrigeren WCET für alle zu übersetzenden Programme widerspiegeln soll. Bevor eine Bewertung der in Kapitel 8 aufgezeigten Ergebnisse in Abschnitt 9.2 gegeben wird, soll zunächst im folgenden Abschnitt eine Zusammenfassung der vorhergehenden Kapitel dargeboten werden.

9.1 Zusammenfassung

Die Einleitung in Kapitel 1 führt den Begriff **Eingebetteter Echtzeitsysteme** ein und zeigt, dass die **WCET** von besonderer Bedeutung für diese ist. Dieser Wert stellt auch die hauptsächliche Größe im Rahmen dieser Arbeit dar, weshalb auch die Entwicklung einer **Registerallokation** motiviert wird, die sich der Minimierung der WCET gezielt widmet. Insbesondere erläutert Abschnitt 1.2 die **Zielstellung** dieser Diplomarbeit. Den Abschluss dieses Kapitels bildet eine Übersicht **verwandter Arbeiten** aus dem Bereich der Registerallokationsverfahren und WCET-optimierenden Methoden.

Das zweite Kapitel stellt eine Reihe verschiedener Begriffe und Definitionen vor, die dem Verständnis dieser Arbeit dienen sollen. Herauszustellen sind hier unter anderen die **WCET**, **WCET-Analyse** und **Registerallokation**, die hier detailliert geschildert werden.

Im Kapitel 3 wird der WCET-optimierende Compiler **WCC** des Lehrstuhls 12 präsentiert. Nachdem seine grundlegende Struktur skizziert wurde, werden im Besonderen die im WCC vorhandenen Registerallokationsverfahren und die WCET-Analyse mittels des **Analyseprogrammes a^3** beschrieben.

Die für diese Arbeit verwendete **Zielarchitektur TC1796** der Firma Infineon wird ausführlich in Kapitel 4 erläutert. Nacheinander werden der grobe Aufbau der Architektur, **Prozessorpipeline**, des **Registersatzes** und **Befehlssatzes** aufgeführt. Der letzte Abschnitt dieses Kapitels beschreibt das spezielle **Verhalten** der Prozessorpipeline und einige Sonderfälle, die für diese Arbeit im weiteren Verlauf eine große Bedeutung haben.

Das bisherige Verfahren der WCET-optimierenden, ILP-basierten **Registerallokation von Schmoll** wird innerhalb des fünften Kapitels umschrieben. Dieses Verfahren basiert auf dem ILP-basierten Ansatz von **Goodwin und Wilken** und verwendet als WCEP-Modellierung die Formulierung von **Suhenrda et al.**, die Schmoll entsprechend erweitert. Beide Verfahren finden daher hier Erwähnung. Außerdem wird gezeigt, wie die $\mathrm{WCET}_{\mathrm{est}}$-Werte aus der statischen WCET-Analyse vom **Spillcode-Overhead** bereinigt werden. Das Ende dieses Kapitels bietet eine Übersicht über das aufgeführte Verfahren. Hier werden auch die **Schwachstellen** des Gesamtverfahrens aufgelistet. Der letzte Abschnitt verweist darauf, wie diese Vorschläge aufgegriffen und in den folgenden Kapiteln behandelt werden.

Kapitel 6 befasst sich mit dem Hauptteil dieser Diplomarbeit, den beiden Pipelineanalyseverfahren. Zunächst wird die Motivation für deren Implementierung und im Anschluss das grundlegende Prinzip beider Verfahren veranschaulicht, von denen zuerst die **statische Pipelineanalyse** vorgestellt wird. Die einführend formulierten ILP-Erweiterungen bilden eine Grundlage für die Modellierung der **dynamischen Pipelineanalyse**. Die Integration beider Verfahren in die bestehende Struktur des Registerallokators findet hier ein besonderes Augenmerk.

Mit der Einbeziehung verschiedener kleinerer Maßnahmen in die Registerallokation zur Steigerung der Güte beschäftigt sich Kapitel 7. So wird der Programmcode für die Dauer der statischen WCET-Analyse dem **SPM** zugeordnet. Des Weiteren erfolgt ein Austausch des naiven **Vorallokationsverfahrens** durch andere Registerallokatoren wie z. B. den RAGC. Darüber hinaus wird die **Copy-Elimination** in die ILP-Modellierung integriert. Zum Abschluss des Kapitels werden zwei Ansätze einer **kombinierten Zielfunktion** vorgestellt, die auch codegrößenoptimierende Elemente berücksichtigen.

Kapitel 8 präsentiert die Ergebnisse des **Benchmarkings**. Eingangs werden die verwendeten Messverfahren beschrieben. Im Anschluss werden die Implementierungen aus den Kapiteln 6 und 7 auf ihre Güte hin untersucht. Diese ist vorrangig das Ergebnis aus den Vergleichen der resultierenden **WCET** mit dem Ergebnis anderer Allokatoren und der Grundversion des Registerallokators zu Beginn dieser Arbeit. Darüber hinaus wird auch die **Laufzeit** der Verfahren gegenübergestellt. Die Auswirkung dieser Veränderungen auf die **ACET** finden in diesem Kapitel Beachtung.

9.2 Bewertung

Nachdem der Inhalt der bisherigen Arbeit noch einmal zusammengefasst ist, soll nun auf die erzielten Ergebnisse aus Kapitel 8 genauer Bezug genommen werden. Zudem soll geklärt werden, ob die Zielsetzung aus Abschnitt 1.2 im Rahmen dieser Diplomarbeit umgesetzt ist.

Zu Anfang dieser Arbeit liegt die Güte des RAILP-WCET bzgl. der resultierenden WCET unter der des RAILP-CS, obwohl dieser nicht die Minimierung der WCET zum Ziel hat. Dieser Zustand ist nicht zufriedenstellend und soll daher beseitigt werden. Vorrangiges Ziel dieser Arbeit ist daher die Steigerung der Güte des RAILP-WCET.

In einem ersten Schritt wird für die Dauer der statischen WCET-Analyse mittels des Analyseprogramms a^3 der Programmcode dem Scratchpad-Speicher zugeordnet. Durch diese Maßnahme entfallen Effekte, die der bisher benutzte Flash-Speicher verursacht, und die Güte der ermittelten WCET_{est}-Werte kann verbessert werden (Diagramm 8.2). Auch der Austausch der Vorallokation steigert die Güte der WCET_{est}-Werte weiter (Diagramm 8.3). Ferner wird auch eine statische Pipelineanalyse implementiert, um die Kosten des in die Vorallokation eingefügten Spillcodes genauer bestimmen zu können. Mit Hilfe aller drei Maßnahmen können die bereinigten Kosten der Basisblöcke genauer bestimmt werden, was die Präzision des ILP-Modells erhöht. Dies entspricht dem ersten Punkt der Zielstellung, der somit umgesetzt ist.

Auch der zweite Punkt der Zielstellung wird erfüllt, da das Verfahren der dynamischen Pipelineanalyse die Kosten für jede einzelne Spillentscheidungen individuell modelliert. Dabei berücksichtigt dieses Verfahren explizit die Eigenschaften der verwendeten Zielarchitektur TC1796. Das Verhalten der zugehörigen Prozessorpipeline wird innerhalb der Modellierung umgesetzt, wobei für die jeweilige Spillentscheidung der Kontext beachtet werden muss, in den die zugehörige Spillinstruktion eingefügt wird. Insbesondere werden auch Spillentscheidungen miteinbezogen, die zu anderen Spillinstruktionen gehören, für die zur Modellierungszeit nicht feststeht, ob sie überhaupt eingefügt werden.

Der dritte und letzte Punkt der Zielstellung sieht vor, dass zusätzliche Erweiterungen implementiert werden, um den RAILP-WCET zu optimieren. Eine solche Erweiterung stellt die Copy-Elimination dar, die sowohl in den RAILP-WCET als auch in den RAILP-CS integriert wird. Für beide Verfahren kann so die Güte der resultierenden WCET merkbar gesteigert werden (Diagramm 8.4). Der größte Zuwachs bzgl. der Güte kann jedoch durch eine Kombination der Zielfunktion erreicht werden. Die hier erzielten Ergebnisse übertreffen die Ergebnisse der ursprünglichen Verfahren deutlich (Abschnitt 8.4). Somit werden zwei Erweiterungen umgesetzt, die auch den letzten Punkt der Zielstellung erfüllen.

Alle Erweiterungen zusammengenommen, können in dieser Diplomarbeit deutliche Verbesserungen bzgl. der resultierenden WCET erreicht werden. Erzielen die ursprünglichen Verfahren RAILP-CS bzw. RAILP-WCET durchschnittliche Verbes-

serungen von 14,9 % bzw. 13,2 % gegenüber dem RAGC, so erreichen die kombinierten Ansätze bis zu 20,3 % bzw. 19,6 %. Besonders herauszustellen ist, dass der Ansatz der ersten Kombination die höchste durchschnittliche Verbesserung aller betrachteten Verfahren liefert. Vor allem übertrifft er den Registerallokator RAILP-CS, obwohl dieser durch die zusätzliche Integration der Copy-Elimination noch beachtlich in seiner Güte gesteigert werden kann. Dabei bezieht die erste Kombination vorrangig die Reduzierung des WCEP als primäres Optimierungsziel in die Modellierung der Zielfunktion mit ein.

Als durchaus positiv können auch die Tatsachen vermerkt werden, dass sowohl die resultierende Laufzeit als auch die ACET für die kombinierten Ansätze nur geringfügig über den Werten des RAILP-WCET zu Beginn dieser Arbeit liegen.

In den Prozess der Registerallokation schließen beide kombinierten Ansätze die beiden Pipelineanalyseverfahren ein, die im Rahmen dieser Diplomarbeit entwickelt werden. Diese berücksichtigen das komplexe Verhalten der Zielarchitektur TC1796. Trotzdem ist es nicht möglich für alle Benchmarks ausschließlich Verbesserungen zu erzielen. Nachfolgend sollen hierfür einige mögliche Gründe genannt werden.

- Es wird versucht das Pipelineverhalten möglichst genau nachzuempfinden, aber aufgrund der Komplexität sind einige Einschränkungen notwendig. So wird die Möglichkeit, dass IP-Befehle die Execute-Stufe innerhalb der Wiederholungsrate durchlaufen, in der ILP-Modellierung nicht berücksichtigt. Des Weiteren muss zur Bestimmung bzw. Modellierung der Kosten einer Spillinstruktion in der statischen bzw. dynamischen Pipelineanalyse das Umfeld der Instruktion ausgewertet werden. Nun kann es sein, dass Befehle am Anfang eines Basisblocks mehrere Vorgänger besitzen. In diesem Fall findet keine weitere Auswertung statt.
- Im Rahmen der statischen Pipelineanalyse werden die Kosten der eingefügten Spillinstruktionen innerhalb der Vorallokation genau bestimmt. Diese werden jedoch von einem $\mathrm{WCET}_{\mathrm{est}}$-Wert abgezogen, der aus einer statischen WCET-Analyse stammt. Die WCET-Analyse ist mit einer gewissen Ungenauigkeit verbunden, da durch eine Überapproximaion sichergestellt werden soll, dass das Resultat die reale WCET keinesfalls überschreitet. Ebendiese Ungenauigkeit überträgt sich auch in die Modellierung des WCEP.
- In der Modellierung des WCEP wird darüber hinaus die Vereinfachung gemacht, dass nicht zwischen den verschiedenen Ausführungskontexten unterschieden wird, die eine Instruktion oder ein Basisblock besitzen kann. Stattdessen wird immer der Worst-Case angenommen und im weiteren Verlauf verwendet.
- In der Hoffnung die Präzision der WCEP-Modellierung zu steigern, wird der Programmcode für die Dauer der WCET-Analyse dem Scratchpad-Speicher zugeordnet. Auf diese Weise sollen die Effekte des Flash-Speichers, wie in Abschnitt 7.1 beschrieben, ausgeschlossen werden. Im Bereich der WCET-Analyse zeigt diese Maßnahme die gewünschten Effekte. Wird der fertige Programmco-

de jedoch während der Ausführung im Flash-Speicher gehalten, treten wieder jene Effekte zu Tage und ein Verhalten resultiert, das in der ILP-Formulierung keinerlei Beachtung findet.

- Außerdem besteht die Möglichkeit, dass mehrere Lösungen für ein ILP existieren, die die Zielfunktion minimieren. Der ILP-Solver CPLEX gibt dennoch nur eine Lösung aus, die er nach eigenen Kriterien auswählt. Die verschiedenen Lösungen können durchaus einen Unterschied bzgl. der Güte der resultierenden WCET verursachen. Durch die kombinierten Ansätze wird eine stärkere Einschränkung des Lösungsraums versucht, um so möglichst die Rückgabe der besten Lösung zu forcieren. In wie weit dies von Erfolg getragen ist, kann jedoch schwer beurteilt werden.

Am Schluss dieses Abschnittes drängt sich nunmehr die Frage auf, ob so viel Aufwand für eine Registerallokation gerechtfertigt scheint.

Hinsichtlich der Bedeutung der WCET im Bereich Eingebetteter Systeme ist eine derart minutiös gestaltete Registerallokation sicherlich gerechtfertigt, wenn sie die entsprechenden Verbesserungen erbringt – gerade in Anbetracht des hohen Optimierungspotenzials der Registerallokation. Speziell die erste Kombination hat mit einer durchschnittlichen Verbesserung von 20,3 % gegenüber dem graphfärbungsbasiertem Registerallokator die Erwartungen erfüllt. Auf der anderen Seite erreicht auch die zweite Kombination vergleichbare Werte und das, obwohl sie eher heuristisch verfährt und weitaus weniger Aufwand betreibt.

„The future is embedded,
embedded is the future!“

Peter Marwedel
[Mar09, Folie 4]

10 Ausblick

Viele der von Schmoll vorgeschlagenen Verbesserungsansätze (Abschnitt 5.4) wurden neben anderen Erweiterungen in dieser Diplomarbeit umgesetzt. Die Umsetzung des Ansatzes von Fu und Wilken [FW02], wie Schmoll sie vorschlägt, wird jedoch nicht realisiert. Durch die Implementierung kann die Rechenzeit beachtlich reduziert werden, was Fu und Wilken auch zeigen. Das Verfahren der Rematerialization, wie es in [GW96] beschrieben wird, kann ebenso zusätzlich implementiert werden, wie es in dieser Arbeit mit der Copy-Elimination geschieht. Bei beiden möglichen Erweiterungen müssen jedoch die Eigenschaften der verwendeten Prozessorarchitektur beachtet werden.

Die WCET-Werte der verschiedenen Ausführungskontexte eines Basisblocks können voneinander abweichen. Einfach die Ausführungsanzahl mit dem Worst-Case zu multiplizieren, wie es bisher geschieht, kann zu einer Ungenauigkeit in der Modellierung des WCEP führen, die eine optimale Registerallokation verhindert. Deshalb sollten in der Modellierung des WCEP alle Ausführungskontexte für die einzelnen Basisblöcke berücksichtigt werden, was in einer genaueren Modellierung des WCEP und besseren Ergebnissen resultieren würde.

Zur Bestimmung der $\mathrm{WCET}_{\mathrm{est}}$-Werte aller Basisblöcke, die zur WCEP-Modellierung benötigt werden, muss bislang eine Vorallokation durchgeführt werden, in der Spillcode eingefügt wird. Dieser wiederum muss nachträglich aus den resultierenden Kosten der statischen WCET-Analyse herausgerechnet werden. Eine sinnvolle Erweiterung stellt die Integration einer statischen WCET-Analyse dar, die die Kosten von Spillinstruktionen nicht beachtet. Das bisher verwendete Analyseprogramm a^3 ist dazu nicht imstande. Durch das Ignorieren der Spillinstruktionen müsste der Spillcode-Overhead nicht mehr über einen Umweg mittels Schätzverfahren oder der statischen Pipelineanalyse approximiert werden. Diese Veränderung kann zu genaueren $\mathrm{WCET}_{\mathrm{est}}$-Werten führen und somit auch eine präzisere Modellierung einhergehend mit besseren Ergebnissen ermöglichen.

Die bisherige Implementierung beschränkt sich auf die Erzeugung einer Registerallokation für den TC1796 und sieht keine Retargierbarkeit vor. Daher scheint es sinnvoll, vor allem die Modellierung von individuellen Kosten für jede einzelne Spillinstruktion innerhalb des ILPs auch auf andere Architekturen anwenden zu können. Dies setzt ein präzises Wissen über das Verhalten der jeweils verwendeten Prozessorpipeline voraus.

Als letztes soll an dieser Stelle noch einer Erweiterung der Formel 7.5 vorgeschlagen werden. Das zugehörige Verfahren stellt wie schon beschrieben einen heuristischen Ansatz dar, der codegrößenoptimierende Elemente mit den individuellen Kosten der Spillinstruktionen innerhalb der Zielfunktion verknüpft, wobei keine Eigenschaften des WCEPs berücksichtigt werden. Die mögliche Ausführungsanzahl einer Spillinstruktion kann jedoch zusätzlich in die Berechnung der Gesamtkosten miteinbezogen werden. Das Verfahren würde so weiterhin ohne eine statische Analyse von WCET-Werten auskommen.

A ILP-Formulierung des Größer-als

Im Folgenden wird eine ILP-Formulierung hergeleitet, die ein Größer-als zwischen einer Integer-Variable b und einem konstanten, ganzzahligen Wert a modelliert. Das Ergebnis dieser Vergleichoperation wird auf die binäre Entscheidungsvariable x abgebildet.

Es soll gelten, dass x genau dann 1 ist, wenn b größer als a ist.

(1) $b > a \implies x = 1$
(2) $x = 1 \implies b > a$

Nach Umformung der Implikationen und Ersetzung des Größer-Operators folgt:

(1) $b \leq a \vee x = 1$
(2) $x \neq 1 \vee b \geq a + 1$

Da x eine binäre Entscheidungsvariable ist, kann die zweite Zeile vereinfacht werden:

(1) $b \leq a \vee x = 1$
(2) $x = 0 \vee b \geq a + 1$

Nach [Bis09] kann die Disjunktion für beide Zeilen auf folgende Weise modelliert werden. Die drei Formeln einer Zeile sind so modelliert, dass sie zu allen Zeitpunkten erfüllt sind. Dies ist nötig, da in einem Optimierungsproblem zu jedem Zeitpunkt alle Nebenbedingungen erfüllt sein müssen.

(1) $b \leq a + Lz_1$	$x \leq 1 + (1 - z_1)$	$x \geq 1 - (1 - z_1)$
(2) $x \leq 0 + z_2$	$x \geq 0 - z_2$	$b \geq (a+1) - M(1 - z_2)$

Dabei gilt: $L = M = \infty$, z_1 und z_2 sind binäre Entscheidungsvariablen

Da x eine binäre Entscheidungsvariable ist, sind die Formeln $x \leq 1+(1-z_1)$ und $x \geq 0 - z_2$ immer erfüllt und können daher weggelassen werden. Unter Berücksichtigung dieser Eigenschaften erhält man:

(1) $b \leq a + Lz_1$ $\quad x \geq 1 - (1 - z_1)$
(2) $x \leq z_2$ $\quad b \geq (a+1) - L(1 - z_2)$

Die Entscheidungsvariablen z_1 und z_2 müssen gleich sein, da anderenfalls Konstellationen entstehen können, bei denen nicht alle Formeln der beiden Zeilen erfüllt sind. Es folgt daher, dass $z = z_1 = z_2$ ist.

(1) $b \leq a + Lz$ $\quad x \geq z$
(2) $x \leq z$ $\quad b \geq (a + 1) - L(1 - z)$

Des Weiteren kann man erkennen, dass wegen der beiden Formeln $x \geq z$ und $x \leq z$ x gleich z sein muss.

(1) $b \leq a + Lx$
(2) $b \geq (a + 1) - L(1 - x)$

Mit diesen beiden Formeln lässt sich nun ein Größer-als, das auf eine binäre Entscheidungsvariable abgebildet wird, in einem ILP modellieren. Analog ist auch die Modellierung des Kleiner-als möglich.

Durch Betrachtung der Formeln kann deren Korrektheit geschlussfolgert werden. Ist b größer als a, so nimmt x den Wert Eins an und es gilt:

(1) $b \leq \infty$
(2) $b \geq (a + 1)$

Falls jedoch b kleiner oder gleich a ist, so nimmt x den Wert Null an und es gilt:

(1) $b \leq a$
(2) $b \geq -\infty$

B Verwendete Benchmarks

Tabelle B.1: Verwendete Benchmarks

MediaBench-Benchmark-Suite [LPMS97]		
cjpeg_jpeg6b_transupp	cjpeg_jpeg6b_wrbmp	h264dec_ldecode_block
MiBench-Benchmark-Suite [GRE+01]		
sha		
MTRC-Benchmark-Suite [EG06]		
adpcm_decoder	expint	matmult
adpcm_encoder	fdct	minver
binarysearch	fibcall	ndes
bsort100	fir	prime
compressdata	insertsort	qsort-exam
countnegative	janne_complex	qurt
cover	jfdctint	select
crc	lms	sqrt
edn	ludcmp	st
UTDSP-Benchmark-Suite [Lee98]		
fir_32_1	latnrm_8_1	qmf_receive
fir_256_64	latnrm_32_64	qmf_transmit
g721.marcuslee_decoder	lmsfir_8_1	v32.modem_achop
g721.marcuslee_encoder	lmsfir_32_64	v32.modem_cnoise
histogram	lpc	v32.modem_eglue
iir_1_1	mult_4_4	
iir_4_64	mult_10_10	
ohne Benchmark-Suite-Zugehörigkeit		
codecs_dcodhuff	hamming_window	selection_sort
h263	searchmultiarray	

C Messreihen

Messreihe I: RAGC, RAILP-CS, RAGC-WCET

- ohne Copy-Elimination für RAILP-CS

Messreihe II: RAILP-WCET

- RASA als Vorallokation
- Programmcode während der Analyse der Vorallokation im Flash-Speicher
- ohne Copy-Elimination
- einfache Kostenschätzung nach Formel 5.11 und statische Kostenmodellierung nach Formel 5.10
- keine Kombination mit codegrößenoptimierenden Elementen

Messreihe III: RAILP-WCET

- RASA, RAGC, RAILP-CS und RAGC-WCET als Vorallokation
- Programmcode während der Analyse der Vorallokation im SPM
- ohne Copy-Elimination
- einfache Kostenschätzung nach Formel 5.11 und statische Kostenmodellierung nach Formel 5.10
- keine Kombination mit codegrößenoptimierenden Elementen

Messreihe IV: RAILP-CS

- mit Copy-Elimination

Messreihe V: RAILP-WCET

- RAGC als Vorallokation
- Programmcode während der Analyse der Vorallokation im SPM
- mit Copy-Elimination
- einfache Kostenschätzung nach Formel 5.11 und statische Kostenmodellierung nach Formel 5.10
- keine Kombination mit codegrößenoptimierenden Elementen

Messreihe VI: RAILP-WCET

- RAGC als Vorallokation
- Programmcode während der Analyse der Vorallokation im SPM
- mit Copy-Elimination
- statische Pipelineanalyse nach Formel 6.1 und dynamische Pipelinenanalyse nach Formel 6.21
- keine Kombination mit codegrößenoptimierenden Elementen

Messreihe VII: RAILP-WCET

- RAGC als Vorallokation
- Programmcode während der Analyse der Vorallokation im SPM
- mit Copy-Elimination
- statische Pipelineanalyse nach Formel 6.1 und dynamische Pipelinenanalyse nach Formel 6.21
- Kombination mit codegrößenoptimierenden Elementen in den Verhältnissen 70:30, 80:20, 90:10 und 99:1 für die Faktoren a und b nach Formel 7.6

Messreihe VIII: RAILP-WCET

- mit Copy-Elimination
- Kombination mit codegrößenoptimierenden Elementen in den Verhältnissen 1:99, 20:80, 50:50, 80:20 und 99:1 für die Faktoren a und b nach Formel 7.6

Anmerkung: Alle Unterstreichungen innerhalb der nachfolgenden Messwertetabellen geben an, dass für die zugehörige Einzelmessung ein Timeout bei der Lösung des ILPs aufgetreten ist. Ein angegebener WCET-Wert entspricht der bis dahin besten gefundenen Lösung des ILPs, die der ILP-Solver zurückgegeben hat.

D Messwerte 1

Tabelle D.1: WCET-Werte der Messreihen I und II [Zyklen]

Benchmark	RAGC	RAILP-CS	RAGC-WCET	RAILP-WCET
adpcm_decoder	630.810	907.493	629.864	698.860
adpcm_encoder	632.536	911.275	632.730	694.057
binarysearch	394	391	350	413
bsort100	298.466	288.566	298.525	293.515
cjpeg_jpeg6b_transupp	439.355.505	61.403.808	168.889.029	57.534.081
cjpeg_jpeg6b_wrbmp	360.217	298.171	319.271	523.188
codecs_dcodhuff	2.387.952	2.192.508	1.951.427	1.960.867
compressdata	2.785	2.849	2.652	2.887
countnegative	30.969	33.547	30.316	31.367
cover	44.973	43.364	44.973	44.858
crc	176.948	182.147	161.044	204.122
edn	113.728	121.767	101.723	111.142
expint	22.039	9.072	51.712	12.644
fdct	10.592	3.578	8.190	3.458
fibcall	1.071	1.071	1.071	1.071
fir	9.095	9.159	4.627	7.823
fir_256_64	19.525	16.705	15.182	19.506
fir_32_1	968	928	964	967
g721.marcuslee_decoder	245.557	245.556	231.114	296.110
g721.marcuslee_encoder	310.551	469.376	296.130	469.376
h263	11.799.169	4.197.550	6.950.121	4.319.213
h264dec_ldecode_block	213.885	72.999	103.075	128.817
hamming_window	39.330	23.158	24.168	23.155
histogram	589.268	308.138	322.097	318.535
iir_1_1	162	137	152	134
iir_4_64	72.277	38.521	41.034	39.415
insertsort	3.398	2.975	3.170	2.984
janne_complex	958	958	967	961
jfdctint	14.810	5.550	11.715	5.403
latnrm_32_64	329.556	194.623	261.827	200.641
latnrm_8_1	1.164	1.030	1.024	1.092
lms	1.735.506	1.698.846	1.786.447	1.728.718
lmsfir_32_64	449.617	368.056	331.517	452.552
lmsfir_8_1	616	611	542	414
lpc	1.068.485	1.073.208	1.033.253	985.848
ludcmp	10.059	7.221	8.838	7.506
matmult	504.133	370.699	356.797	362.079
minver	8.264	5.859	6.553	6.138
mult_10_10	64.247	36.223	40.564	35.024
mult_4_4	2.652	2.177	2.232	2.223
ndes	211.342	204.839	215.646	190.475
prime	20.385	17.403	20.361	21.685
qmf_receive	9.667.831	4.254.911	7.257.229	4.066.029
qmf_transmit	10.417.289	4.068.669	9.216.556	3.995.012
qsort-exam	41.523	36.494	41.277	39.890
qurt	8.784	10.164	9.102	8.530
searchmultiarray	38.697	38.589	38.697	52.485
select	17.078	16.337	10.384	16.769
selection_sort	5.883.641	5.880.681	4.678.557	5.965.616
sha	10.779.577	8.483.523	19.036.316	8.478.004
sqrt	14.354	15.932	14.559	16.750
st	499.498	508.112	504.485	498.503
v32.modem_achop	17.869	18.109	17.964	17.629
v32.modem_cnoise	169.463	168.492	166.096	159.836
v32.modem_eglue	23.465	23.465	23.465	24.899

Tabelle D.2: Laufzeiten der Messreihen I und II [hh:mm:ss.ff]

Benchmark	RAGC	RAILP-CS	RAGC-WCET	RAILP-WCET
adpcm_decoder	00.74	07.89	05:46.35	05:54.76
adpcm_encoder	00.82	08.43	07:14.59	06:12.09
binarysearch	00.02	00.62	04.62	01.30
bsort100	00.02	00.91	51.50	01:11.89
cjpeg_jpeg6b_transupp	00.91	01:10:21.95	22:09.36	23:44.40
cjpeg_jpeg6b_wrbmp	00.25	03.84	01:22:04.12	17:01.58
codecs_dcodhuff	00.14	04.16	02:26.08	32.16
compressdata	00.04	01.31	16.98	03.33
countnegative	00.05	01.73	16.53	10.75
cover	00.17	39.81	04:36.60	04:13.41
crc	00.13	02.19	01:07.68	40.39
edn	00.79	14.14	04:32.96	01:24.43
expint	00.06	01.89	15.65	06.63
fdct	00.82	40:07.31	28.02	12:10.14
fibcall	00.01	00.29	01.97	00.84
fir	00.04	01.12	06.15	03.01
fir_256_64	00.02	00.66	06.08	03.57
fir_32_1	00.01	00.51	02.71	01.26
g721.marcuslee_decoder	00.02	01.05	03.45	01.72
g721.marcuslee_encoder	00.03	01.92	07.40	05.20
h263	00.34	09.09	08:44.15	28:27.37
h264dec_ldecode_block	03.49	50.64	30:13.02	07:25.84
hamming_window	00.03	00.73	09.96	07.55
histogram	00.07	01.74	40.80	27.78
iir_1_1	00.02	00.38	01.67	00.77
iir_4_64	00.08	01.69	14.50	10.30
insertsort	00.02	00.63	03.98	01.85
janne_complex	00.01	00.48	03.82	01.17
jfdctint	00.64	40:05.75	30.97	40:14.77
latnrm_32_64	00.09	05.61	38.81	33.76
latnrm_8_1	00.04	00.95	05.37	01.78
lms	00.23	04.75	04:26.80	04:16.99
lmsfir_32_64	00.06	01.84	51.42	59.36
lmsfir_8_1	00.03	00.98	05.36	01.74
lpc	01.70	12.64	15:28.27	04:00.72
ludcmp	00.26	04.03	01:20.84	10.01
matmult	00.06	01.55	01:14.95	01:21.40
minver	00.31	06.79	01:30.41	16.39
mult_10_10	00.03	00.87	10.52	07.55
mult_4_4	00.04	00.83	04.86	02.77
ndes	00.54	06.76	05:22.92	01:22.20
prime	00.02	00.55	06.95	05.25
qmf_receive	00.11	03.85	02:03.45	02:37.21
qmf_transmit	00.12	03.49	02:15.14	01:58.15
qsort-exam	00.26	04.11	15.69	16.55
qurt	00.08	01.96	23.18	04.72
searchmultiarray	00.02	00.84	08.48	15.05
select	00.36	04.55	48.20	12.35
selection_sort	00.02	00.77	01:42.27	01:43.47
sha	00.51	12.92	16:52.41	03:34.07
sqrt	00.03	01.08	12.04	03.84
st	00.17	04.99	02:04.92	01:44.45
v32.modem_achop	00.01	00.48	05.35	04.00
v32.modem_cnoise	00.08	01.53	13.94	02.61
v32.modem_eglue	00.01	00.36	04.46	04.52
Durchschnitt	00.27	02:48.76	04:13.07	03:12.75

E Messwerte 2

Tabelle E.1: WCET-Werte der Messreihe III [Zyklen]

Benchmark	Vorallokation RASA	Vorallokation RAGC	Vorallokation RAILP-CS	Vorallokation RAGC-WCET
adpcm_decoder	698.906	698.971	698.951	698.971
adpcm_encoder	701.592	701.497	701.489	701.497
binarysearch	390	405	417	405
bsort100	287.577	288.565	288.565	288.565
cjpeg_jpeg6b_transupp	61.238.960	53.863.815	40.568.388	58.185.221
cjpeg_jpeg6b_wrbmp	445.807	354.810	320.275	341.449
codecs_dcodhuff	2.281.720	2.294.976	2.294.976	2.411.101
compressdata	2.806	2.796	2.796	2.796
countnegative	31.181	33.047	31.010	33.047
cover	46.031	44.928	44.919	44.920
crc	192.457	186.654	186.654	186.654
edn	111.142	111.142	111.142	111.142
expint	11.161	11.056	9.409	9.962
fdct	3.458	3.458	3.458	3.458
fibcall	1.071	1.071	1.071	1.071
fir	7.715	7.765	7.744	7.694
fir_256_64	19.760	18.229	18.229	18.229
fir_32_1	967	967	967	967
g721.marcuslee_decoder	233.527	296.109	296.109	296.109
g721.marcuslee_encoder	421.268	380.337	380.337	380.337
h263	4.376.807	4.204.592	3.995.965	4.003.799
h264dec_ldecode_block	120.835	129.237	80.720	80.666
hamming_window	23.155	23.155	23.155	23.155
histogram	318.535	318.535	318.535	318.535
iir_1_1	134	134	134	134
iir_4_64	39.415	39.415	39.415	39.415
insertsort	2.983	2.984	2.984	2.984
janne_complex	1.144	950	950	950
jfdctint	5.367	5.367	5.367	5.367
latnrm_32_64	200.641	200.641	200.641	200.641
latnrm_8_1	1.092	1.092	1.092	1.092
lms	1.812.710	1.819.528	1.823.994	1.819.528
lmsfir_32_64	423.418	424.070	465.106	465.106
lmsfir_8_1	414	414	414	414
lpc	1.002.075	952.500	1.016.717	1.000.322
ludcmp	7.740	8.331	7.507	7.394
matmult	362.079	362.079	362.079	362.079
minver	5.379	5.875	5.685	6.049
mult_10_10	35.024	35.024	35.024	35.024
mult_4_4	2.223	2.223	2.223	2.223
ndes	219.517	203.851	195.513	210.455
prime	21.685	21.685	21.685	21.685
qmf_receive	4.204.513	4.433.470	4.026.167	3.903.379
qmf_transmit	3.727.957	4.052.304	4.129.659	4.306.711
qsort-exam	42.488	41.178	41.178	41.178
qurt	8.237	9.430	9.487	9.430
searchmultiarray	40.684	40.684	40.684	40.684
select	16.429	15.316	16.867	15.316
selection_sort	5.957.515	5.957.842	5.957.842	5.957.842
sha	8.657.981	8.497.440	8.466.456	8.712.189
sqrt	14.932	14.911	14.911	14.911
st	512.500	532.520	511.988	532.520
v32.modem_achop	17.629	17.629	17.629	17.629
v32.modem_cnoise	164.650	164.650	164.650	164.650
v32.modem_eglue	24.899	24.899	24.899	24.899

Tabelle E.2: Laufzeiten der Messreihe III [hh:mm:ss.ff]

Benchmark	Vorallokation RASA	Vorallokation RAGC	Vorallokation RAILP-CS	Vorallokation RAGC-WCET
adpcm_decoder	06:02.24	56.73	01:50.64	06:36.35
adpcm_encoder	06:05.84	58.27	01:52.66	08:02.43
binarysearch	01.28	01.02	01.52	05.67
bsort100	01:10.30	23.60	24.65	01:14.35
cjpeg_jpeg6b_transupp	38:56.38	44:56.85	01:24:04.62	59:48.61
cjpeg_jpeg6b_wrbmp	17:41.66	48.70	09.48	01:16:09.40
codecs_dcodhuff	27.16	10.38	14.18	02:37.06
compressdata	03.34	01.97	03.19	17.18
countnegative	10.54	04.12	05.63	19.73
cover	05:54.11	05:26.64	06:06.60	10:08.49
crc	40.27	13.63	15.86	01:07.12
edn	01:24.20	29.92	42.07	04:54.73
expint	06.76	07.17	06.78	23.01
fdct	11:14.11	05:06.08	42:28.65	05:21.13
fft_1024	01:57.63	02:21.63	16.46	01:28.50
fft_256	46.11	54.33	09.27	43.53
fibcall	00.89	00.60	00.90	02.49
fir	03.15	02.28	03.16	08.26
fir_256_64	03.39	01.64	02.25	07.93
fir_32_1	01.25	00.82	01.22	03.52
g721.marcuslee_decoder	01.60	01.42	02.35	04.96
g721.marcuslee_encoder	05.26	02.81	04.42	10.25
h263	20:58.81	10:45.44	01:28.56	14:18.08
h264dec_ldecode_block	13:02.64	13:08.31	09:32.21	29:50.12
hamming_window	07.39	04.44	03.80	12.88
histogram	27.52	21.67	10.68	51.77
iir_1_1	00.75	00.60	00.94	02.32
iir_4_64	10.13	06.77	04.80	18.98
insertsort	01.88	01.07	01.65	04.82
janne_complex	01.21	00.90	01.37	04.70
jfdctint	40:12.95	40:08.72	01:20:11.25	40:38.32
latnrm_32_64	33.29	21.38	14.71	52.65
latnrm_8_1	01.79	01.30	02.03	06.79
lms	04:12.81	01:10.37	01:15.49	05:31.96
lmsfir_32_64	58.19	23.24	20.93	01:13.21
lmsfir_8_1	01.73	01.23	01.99	06.66
lpc	03:57.32	01:28.45	01:25.50	16:33.52
ludcmp	10.79	09.35	10.85	01:29.21
matmult	01:19.83	41.76	20.80	01:39.35
minver	12.69	10.18	17.79	01:40.33
mult_10_10	07.36	05.09	03.24	13.20
mult_4_4	02.73	01.49	01.99	06.41
ndes	01:20.72	28.78	32.60	05:56.37
prime	05.08	01.97	02.89	08.60
qmf_receive	01:49.18	01:07.35	37.45	02:59.52
qmf_transmit	02:08.72	01:22.17	49.16	03:19.04
qsort-exam	15.65	09.67	13.56	24.62
qurt	04.57	02.87	04.83	25.90
searchmultiarray	14.66	03.45	04.30	11.38
select	13.96	08.30	14.47	46.58
selection_sort	01:38.78	48.63	50.54	02:36.45
sha	02:53.63	55.41	49.09	18:49.88
sqrt	03.90	02.38	03.31	11.05
st	01:44.60	28.29	33.05	02:31.63
v32.modem_achop	03.85	01.53	01.97	06.77
v32.modem_cnoise	04.95	02.00	03.28	16.28
v32.modem_eglue	03.68	01.66	02.02	06.02
Durchschnitt	03:26.94	02:27.11	04:21.34	06:02.69

F Messwerte 3

Tabelle F.1: WCET-Werte und Laufzeiten der Messreihen IV und V

	WCET [Zyklen]		Laufzeit [hh:mm:ss.ff]	
Benchmark	RAILP-CS	RAILP-WCET	RAILP-CS	RAILP-WCET
adpcm_decoder	705.733	814.925	08.70	01:02.22
adpcm_encoder	709.691	817.648	09.33	01:01.73
binarysearch	335	339	00.70	01.03
bsort100	288.564	288.359	00.97	24.21
cjpeg_jpeg6b_transupp	59.171.356	74.265.575	39:09.68	13:24.88
cjpeg_jpeg6b_wrbmp	314.458	283.823	05.66	50.54
codecs_dcodhuff	2.237.137	2.429.738	07.47	15.71
compressdata	2.716	2.753	02.58	05.07
countnegative	33.579	34.001	02.95	05.32
cover	43.212	42.952	35.50	01:45.42
crc	181.638	180.741	04.04	13.79
edn	110.800	109.571	25.88	34.51
expint	7.808	8.102	03.95	07.35
fdct	3.618	3.441	40:03.62	40:12.05
fibcall	1.071	1.071	00.89	01.25
fir	4.395	9.781	01.89	03.38
fir_256_64	15.409	14.133	00.66	01.96
fir_32_1	947	875	00.49	00.83
g721.marcuslee_decoder	235.928	238.335	01.70	01.82
g721.marcuslee_encoder	382.743	469.375	03.87	05.51
h263	3.917.166	4.122.686	14.28	10:54.27
h264dec_ldecode_block	68.385	75.359	14:51.52	14:44.58
hamming_window	25.206	25.187	01.13	05.20
histogram	311.767	316.729	02.90	22.35
iir_1_1	144	138	00.39	01.19
iir_4_64	37.473	37.133	01.53	07.11
insertsort	3.155	3.205	01.15	01.72
janne_complex	844	844	00.58	00.97
jfdctint	5.742	5.529	40:03.99	40:16.84
latnrm_32_64	188.478	208.636	20.58	21.91
latnrm_8_1	960	974	01.43	02.17
lms	1.815.595	1.894.011	06.20	01:12.63
lmsfir_32_64	412.986	415.816	01.98	24.30
lmsfir_8_1	547	565	01.06	01.45
lpc	1.103.742	913.986	01:11.28	01:35.60
ludcmp	8.066	7.064	01:35.67	51.85
matmult	407.345	379.664	02.88	42.75
minver	5.684	5.838	01:28.37	01:33.74
mult_10_10	37.210	37.610	00.92	05.22
mult_4_4	1.889	1.809	01.49	04.70
ndes	173.087	208.062	07.70	31.08
prime	17.398	17.383	00.83	02.34
qmf_receive	3.862.829	4.068.488	04.25	01:13.04
qmf_transmit	4.174.143	4.134.051	04.16	01:16.13
qsort-exam	41.683	38.562	32.49	15.77
qurt	8.285	8.246	02.82	04.09
searchmultiarray	38.679	42.430	00.83	03.86
select	15.934	15.499	03:43.85	01:08.97
selection_sort	5.369.878	8.173.662	01.09	51.79
sha	8.296.915	8.427.282	19.33	37.95
sqrt	15.271	16.264	01.19	02.24
st	490.867	493.591	05.69	28.18
v32.modem_achop	18.107	16.667	00.46	01.52
v32.modem_cnoise	165.606	159.349	02.43	02.37
v32.modem_eglue	23.465	22.270	00.40	01.69
Durchschnitt	-	-	02:39.77	02:33.28

G Messwerte 4

Tabelle G.1: Anzahl entfernter Kopierbefehle mit und ohne Copy-Elimination

		RAILP-CS aus Messreihe I ohne Copy-Elimination		RAILP-CS aus Messreihe IV mit Copy-Elimination		RAILP-WCET aus Messreihe II ohne Copy-Elimination		RAILP-WCET aus Messreihe V mit Copy-Elimination	
Benchmark	Anzahl der Kopierbefehle	entfernte Befehle	Anteil [%]	entfernte Befehle	Anteil [%]	entfernte Befehle	Anteil [%]	entfernte Befehle	Anteil [%]
cjpeg_jpeg6b_transupp	49	5	10,2	34	69,4	11	22,4	38	77,6
cjpeg_jpeg6b_wrbmp	35	4	11,4	11	31,4	12	34,3	22	62,9
h264dec_ldecode_block	160	11	6,9	61	38,1	63	39,4	109	68,1
sha	187	15	8,0	59	31,6	25	13,4	67	35,8
codecs_dcodhuff	71	5	7,0	21	29,6	7	9,9	19	26,8
h263	47	7	14,9	35	74,5	6	12,8	36	76,6
hamming_window	7	0	0,0	5	71,4	2	28,6	5	71,4
searchmultiarray	4	1	25,0	3	75,0	0	0,0	3	75,0
selection_sort	9	2	22,2	6	66,7	2	22,2	7	77,8
adpcm_decoder	148	19	12,8	77	52,0	34	23,0	88	59,5
adpcm_encoder	167	20	12,0	90	53,9	31	18,6	98	58,7
binarysearch	9	1	11,1	7	77,8	0	0,0	7	77,8
bsort100	3	0	0,0	2	66,7	1	33,3	3	100,0
compressdata	14	1	7,1	6	42,9	4	28,6	6	42,9
countnegative	17	2	11,8	10	58,8	2	11,8	9	52,9
cover	12	0	0,0	7	58,3	2	16,7	7	58,3
crc	30	3	10,0	13	43,3	2	6,7	13	43,3
edn	139	12	8,6	44	31,7	15	10,8	47	33,8
expint	12	0	0,0	6	50,0	1	8,3	7	58,3
fdct	8	1	12,5	2	25,0	0	0,0	2	25,0
fibcall	4	0	0,0	0	0,0	0	0,0	0	0,0
fir	18	2	11,1	10	55,6	3	16,7	9	50,0
insertsort	1	0	0,0	0	0,0	0	0,0	0	0,0
janne_complex	10	0	0,0	6	60,0	1	10,0	6	60,0
jfdctint	6	1	16,7	3	50,0	3	50,0	4	66,7
lms	85	6	7,1	36	42,4	13	15,3	44	51,8
ludcmp	40	2	5,0	21	52,5	7	17,5	26	65,0
matmult	21	3	14,3	9	42,9	1	4,8	8	38,1
minver	67	8	11,9	29	43,3	16	23,9	32	47,8
ndes	98	8	8,2	46	46,9	17	17,3	46	46,9
prime	17	2	11,8	8	47,1	5	29,4	9	52,9
qsort-exam	19	3	15,8	8	42,1	2	10,5	9	47,4
qurt	20	3	15,0	14	70,0	4	20,0	15	75,0
select	55	1	1,8	4	7,3	40	72,7	4	7,3
sqrt	12	5	41,7	7	58,3	5	41,7	8	66,7
st	84	4	4,8	31	36,9	28	33,3	44	52,4
fir_32_1	4	0	0,0	4	100,0	1	25,0	4	100,0
fir_256_64	11	2	18,2	8	72,7	4	36,4	9	81,8
g721.marcuslee_decoder	6	1	16,7	3	50,0	0	0,0	3	50,0
g721.marcuslee_encoder	7	0	0,0	2	28,6	1	14,3	2	28,6
histogram	7	0	0,0	6	85,7	1	14,3	6	85,7
iir_1_1	9	1	11,1	5	55,6	4	44,4	6	66,7
iir_4_64	17	2	11,8	11	64,7	3	17,6	12	70,6
latnrm_8_1	24	2	8,3	14	58,3	6	25,0	16	66,7
latnrm_32_64	26	6	23,1	13	50,0	4	15,4	17	65,4
lmsfir_8_1	16	2	12,5	7	43,8	3	18,8	7	43,8
lmsfir_32_64	24	3	12,5	17	70,8	5	20,8	18	75,0
lpc	116	16	13,8	75	64,7	12	10,3	70	60,3
mult_4_4	11	1	9,1	6	54,5	3	27,3	6	54,5
mult_10_10	11	2	18,2	7	63,6	3	27,3	7	63,6
qmf_receive	38	9	23,7	24	63,2	8	21,1	23	60,5
qmf_transmit	40	5	12,5	24	60,0	5	12,5	23	57,5
v32.modem_achop	2	0	0,0	2	100,0	0	0,0	2	100,0
v32.modem_cnoise	16	4	25,0	9	56,3	2	12,5	9	56,3
v32.modem_eglue	3	0	0,0	1	33,3	1	33,3	2	66,7
Zusammen	2073	213	10,3	969	46,7	431	20,8	1099	53,0

H Messwerte 5

Tabelle H.1: WCET-Werte und Laufzeiten der Messreihe VI

Benchmark	WCET [Zyklen]	Laufzeit [hh:mm:ss.ff]
adpcm_decoder	540.862	01:13.97
adpcm_encoder	548.798	01:15.55
binarysearch	341	01.55
bsort100	394.971	25.01
cjpeg_jpeg6b_transupp	65.292.892	49:17.11
cjpeg_jpeg6b_wrbmp	501.222	54.16
codecs_dcodhuff	2.294.003	57.40
compressdata	2.743	09.53
countnegative	41.950	09.49
cover	45.260	05:06.54
crc	202.195	26.25
edn	115.569	02:09.98
expint	10.496	15.11
fdct	3.642	06:21.19
fibcall	1.071	02.73
fir	9.872	13.04
fir_256_64	15.171	02.39
fir_32_1	915	01.22
g721.marcuslee_decoder	257.591	05.30
g721.marcuslee_encoder	322.569	14.16
h263	4.109.361	11:37.11
h264dec_ldecode_block	112.119	57:42.55
hamming_window	25.212	05.61
histogram	300.960	24.16
iir_1_1	147	00.97
iir_4_64	34.384	08.46
insertsort	2.970	04.04
janne_complex	963	01.82
jfdctint	5.518	40:14.46
latnrm_32_64	210.046	01:21.27
latnrm_8_1	1.054	03.16
lms	1.781.079	01:19.30
lmsfir_32_64	499.380	27.72
lmsfir_8_1	564	02.71
lpc	988.600	04:32.44
ludcmp	6.781	10:10.58
matmult	364.709	43.78
minver	5.593	40:27.50
mult_10_10	37.019	05.96
mult_4_4	1.857	04.91
ndes	180.832	40.74
prime	21.731	02.95
qmf_receive	3.923.258	01:21.54
qmf_transmit	4.191.180	01:45.12
qsort-exam	42.172	03:37.02
qurt	9.203	05.84
searchmultiarray	45.721	05.22
select	16.480	33:30.17
selection_sort	6.770.859	53.67
sha	8.468.965	01:08.69
sqrt	15.315	03.73
st	502.072	34.66
v32.modem_achop	18.107	02.25
v32.modem_cnoise	178.599	03.81
v32.modem_eglue	23.465	02.02
Durchschnitt		03:28.68

I Messwerte 6

Tabelle I.1: WCET-Werte der Messreihe VII [Zyklen]

Benchmark	Verhältnis 70:30	Verhältnis 80:20	Verhältnis 90:10	Verhältnis 99:1
adpcm_decoder	836.279	841.689	699.170	843.701
adpcm_encoder	701.083	750.919	701.061	846.495
binarysearch	332	332	332	332
bsort100	292.615	292.615	292.615	292.615
cjpeg_jpeg6b_transupp	45.080.564	69.171.824	68.605.568	59194063
cjpeg_jpeg6b_wrbmp	286.600	300.668	290.735	310.976
codecs_dcodhuff	2.285.304	2.340.422	1.989.061	2.188.449
compressdata	2.699	2.699	2.699	2.699
countnegative	34.857	36.659	36.659	34.858
cover	43.448	43.448	43.448	43.448
crc	180.961	186.534	179.204	179.262
edn	102.638	102.919	102.740	102.784
expint	8.113	10.282	9.209	8.998
fdct	3.527	3.412	3.326	3.492
fibcall	1.071	1.071	1.071	1.043
fir	4.436	4.436	4.436	9.771
fir_256_64	13.624	13.624	13.624	13.624
fir_32_1	963	963	963	963
g721.marcuslee_decoder	235.928	235.929	226.304	226.304
g721.marcuslee_encoder	337.030	337.030	337.030	337.030
h263	4.248.842	3.931.890	4.204.239	3.982.033
h264dec_ldecode_block	77.234	70.411	74.704	76.097
hamming_window	25.966	25.966	25.966	25.966
histogram	298.015	305.191	313.378	313.378
iir_1_1	144	144	144	144
iir_4_64	40.527	36.240	38.930	38.930
insertsort	3.254	3.254	3.254	3.254
janne_complex	844	844	844	844
jfdctint	5.256	5.438	5.610	5.329
latnrm_32_64	190.208	210.048	199.999	188.928
latnrm_8_1	990	990	968	958
lms	1.761.209	1.813.813	1.807.520	1.838.864
lmsfir_32_64	362.956	386.744	386.744	386.744
lmsfir_8_1	594	548	548	548
lpc	1.090.691	1.006.858	959.341	979.094
ludcmp	6.940	7.088	6.956	6.885
matmult	363.660	362.120	351.341	351.661
minver	5.688	5.667	5.541	5.488
mult_10_10	37.310	37.310	37.310	37.310
mult_4_4	2.063	1.793	1.873	1.967
ndes	181.838	202.630	191.954	205.043
prime	17.395	17.395	17.395	17.395
qmf_receive	3.671.453	3.856.806	4.199.782	4.017.209
qmf_transmit	4.073.598	3.888.571	3.965.457	3.869.931
qsort-exam	38.562	41.292	38.751	39.138
qurt	8.369	8.366	8.223	8.205
searchmultiarray	44.540	44.540	44.540	44.540
select	15.646	15.742	10.324	10.172
selection_sort	5.369.879	5.369.879	5.369.879	5.369.879
sha	8.168.399	8.188.039	8.321.232	7.904.203
sqrt	16.309	16.279	17.263	17.263
st	501.378	501.186	497.685	497.685
v32.modem_achop	18.011	18.011	18.011	18.011
v32.modem_cnoise	168.015	161.764	163.206	172.346
v32.modem_eglue	23.465	23.465	23.465	23.465

Tabelle I.2: Laufzeiten der Messreihe VII [hh:mm:ss.ff]

Benchmark	Verhältnis 70:30	Verhältnis 80:20	Verhältnis 90:10	Verhältnis 99:1
adpcm_decoder	01:17.46	01:20.02	01:20.26	01:16.34
adpcm_encoder	01:21.36	01:24.80	01:22.71	01:21.30
binarysearch	01.87	02.05	01.84	01.88
bsort100	25.50	26.68	25.54	25.27
cjpeg_jpeg6b_transupp	24:51.78	53:26.32	54:08.06	53:18.15
cjpeg_jpeg6b_wrbmp	57.53	55.40	55.61	58.58
codecs_dcodhuff	34.06	38.51	37.69	50.74
compressdata	07.83	10.19	12.69	11.09
countnegative	10.44	13.72	12.49	08.94
cover	03:25.59	03:30.83	03:29.04	03:24.10
crc	22.91	19.80	23.51	24.31
edn	58.45	01:10.82	01:13.43	01:07.07
expint	34.83	53.26	20.46	13.09
fdct	03:42.12	03:07.34	02:22.34	05:48.08
fibcall	01.62	02.63	02.57	01.93
fir	06.92	09.28	08.74	08.80
fir_256_64	02.81	02.82	02.86	02.80
fir_32_1	01.51	01.76	01.61	01.54
g721.marcuslee_decoder	04.10	04.09	04.68	04.31
g721.marcuslee_encoder	06.55	06.80	07.70	06.76
h263	11:03.81	11:55.32	11:14.20	11:01.38
h264dec_ldecode_block	22:30.56	43:21.40	44:17.76	42:53.16
hamming_window	06.17	06.26	06.05	06.06
histogram	25.92	25.60	25.68	26.04
iir_1_1	01.09	01.18	01.09	01.09
iir_4_64	09.51	09.62	09.93	09.88
insertsort	02.94	03.13	03.03	03.66
janne_complex	02.05	02.06	02.04	02.05
jfdctint	20:40.74	40:17.13	12:38.79	14:26.20
latnrm_32_64	29.46	38.12	28.94	36.88
latnrm_8_1	03.88	03.92	03.81	03.75
lms	01:22.37	01:22.64	01:22.48	01:25.96
lmsfir_32_64	26.39	26.80	28.08	27.47
lmsfir_8_1	03.53	03.58	03.60	03.65
lpc	02:05.58	02:05.39	02:12.40	02:11.38
ludcmp	04:23.25	04:16.96	03:57.93	04:14.71
matmult	45.58	45.67	45.81	45.65
minver	05:42.87	19:13.09	23:23.14	40:28.10
mult_10_10	06.57	06.68	06.87	06.47
mult_4_4	03.80	04.13	03.56	03.87
ndes	46.73	46.47	46.43	47.15
prime	03.43	03.38	03.36	03.59
qmf_receive	01:19.77	01:18.98	01:23.66	01:20.14
qmf_transmit	01:27.42	01:20.26	01:28.37	01:23.45
qsort-exam	02:07.79	01:53.66	02:09.29	09:08.50
qurt	08.69	08.33	10.18	07.20
searchmultiarray	05.28	05.35	05.14	05.42
select	02:41.52	04:50.89	16:28.99	08:11.85
selection_sort	52.89	55.46	51.81	50.93
sha	01:11.91	01:07.54	01:11.89	01:08.45
sqrt	04.32	04.34	04.48	04.41
st	38.18	38.47	39.83	39.14
v32.modem_achop	02.25	02.35	02.30	02.27
v32.modem_cnoise	04.75	04.94	04.59	04.46
v32.modem_eglue	02.24	02.30	02.28	02.24
Durchschnitt	02:12.52	03:45.79	03:32.72	03:52.93

J Messwerte 7

Tabelle J.1: WCET-Werte der Messreihe VIII [Zyklen]

Benchmark	Verhältnis 1 zu 99	Verhältnis 20 zu 80	Verhältnis 50 zu 50	Verhältnis 80 zu 20	Verhältnis 99 zu 1
adpcm_decoder	698.413	698.413	705.688	683.844	698.390
adpcm_encoder	702.538	702.538	709.813	687.361	702.503
binarysearch	332	332	332	332	332
bsort100	293.534	293.534	293.534	293.534	293.534
cjpeg_jpeg6b_transupp	58.922.485	70.857.532	58.748.608	66.925.524	81.203.409
cjpeg_jpeg6b_wrbmp	313.009	313.009	316.623	316.623	318.563
codecs_dcodhuff	2.306.961	2.192.178	2.204.132	2.253.246	2.219.396
compressdata	2.667	2.667	2.667	2.742	2.811
countnegative	33.979	33.979	33.979	34.857	36.659
cover	43.223	43.223	43.223	43.223	43.223
crc	181.636	181.636	181.636	175.497	177.028
edn	108.579	109.339	108.472	108.738	110.192
expint	9.210	9.210	8.014	8.203	8.102
fdct	3.447	3.569	3.432	3.501	3.392
fibcall	1.071	1.071	1.071	1.071	1.132
fir	4.436	4.436	4.436	4.406	4.404
fir_256_64	15.155	15.155	15.155	15.162	15.162
fir_32_1	883	883	883	883	883
g721.marcuslee_decoder	235.929	235.929	238.334	240.741	238.335
g721.marcuslee_encoder	389.943	389.943	322.568	375.502	469.376
h263	4.069.424	4.069.423	4.015.564	4.183.484	4.103.804
h264dec_ldecode_block	68.048	67.348	74.204	72.392	81.639
hamming_window	25.605	25.605	25.605	25.605	25.605
histogram	300.830	300.830	300.830	300.830	300.830
iir_1_1	144	144	144	144	144
iir_4_64	39.733	39.733	39.733	40.309	38.005
insertsort	3.207	3.207	3.207	3.162	3.198
janne_complex	844	844	844	844	898
jfdctint	5.285	5.339	5.458	5.460	5.111
latnrm_32_64	209.535	200.253	203.327	214.143	214.718
latnrm_8_1	927	927	1.015	950	959
lms	1.753.132	1.753.132	1.668.472	1.751.734	1.791.564
lmsfir_32_64	362.294	362.294	362.294	413.980	413.980
lmsfir_8_1	533	533	533	533	533
lpc	1.077.807	1.069.418	1.082.327	1.079.050	1.081.732
ludcmp	7.944	7.954	7.071	7.100	7.235
matmult	362.941	362.941	362.941	368.039	363.341
minver	5.593	6.240	5.644	5.294	6.311
mult_10_10	37.310	37.310	37.310	41.610	41.610
mult_4_4	1.777	1.777	1.777	1.777	1.777
ndes	177.546	177.546	177.596	175.856	186.640
prime	17.397	17.397	17.397	17.397	17.397
qmf_receive	3.849.969	4.149.473	3.675.077	3.769.794	4.138.004
qmf_transmit	3.827.218	4.157.452	4.336.718	3.672.016	3.940.584
qsort-exam	31.599	41.592	41.442	41.643	39.256
qurt	8.288	8.306	8.294	8.387	8.328
searchmultiarray	38.685	38.685	38.685	38.685	38.685
select	16.539	14.622	16.527	15.214	16.450
selection_sort	8.171.578	8.171.578	8.172.176	5.370.179	5.369.879
sha	8.201.650	8.223.300	8.055.023	8.185.704	8.398.702
sqrt	16.285	16.285	16.285	16.315	16.315
st	494.827	494.827	494.779	492.825	492.825
v32.modem_achop	18.011	18.011	18.011	18.011	18.011
v32.modem_cnoise	158.878	158.878	158.878	158.883	158.883
v32.modem_eglue	23.465	23.465	23.465	23.465	23.465

Tabelle J.2: Laufzeiten der Messreihe VIII [hh:mm:ss.ff]

Benchmark	Verhältnis 1 zu 99	Verhältnis 20 zu 80	Verhältnis 50 zu 50	Verhältnis 80 zu 20	Verhältnis 99 zu 1
adpcm_decoder	28.76	29.66	28.04	30.57	29.64
adpcm_encoder	31.56	32.65	30.59	32.55	34.61
binarysearch	01.34	01.32	01.29	01.54	03.92
bsort100	02.20	02.32	02.36	02.37	02.50
cjpeg_jpeg6b_transupp	01:24:22.96	01:26:40.70	01:24:37.79	01:30:27.20	01:27:05.70
cjpeg_jpeg6b_wrbmp	14.40	14.34	14.73	15.14	15.26
codecs_dcodhuff	22.67	34.43	24.58	37.17	01:04.26
compressdata	05.65	06.10	06.14	08.58	11.70
countnegative	06.27	06.48	06.97	07.51	12.16
cover	02:50.13	02:49.60	03:16.80	02:50.41	03:29.27
crc	08.47	08.28	11.17	21.82	13.38
edn	48.61	47.66	48.24	03:23.02	54.07
expint	09.31	09.20	07.69	01:13.37	54.60
fdct	19:37.74	18:11.57	22:23.31	28:15.74	36:35.80
fibcall	02.53	02.51	01.43	02.42	04.07
fir	04.05	04.03	04.04	05.49	06.44
fir_256_64	01.52	01.56	01.56	01.52	01.60
fir_32_1	01.01	01.04	01.13	01.04	01.01
g721.marcuslee_decoder	04.43	04.45	07.26	06.17	14.35
g721.marcuslee_encoder	08.05	08.36	27.94	09.37	01:09.40
h263	41.21	40.14	33.94	26.56	28.95
h264dec_ldecode_block	36:47.94	42:47.18	20:41.20	55:14.73	48:51.77
hamming_window	02.61	02.57	02.56	02.45	02.63
histogram	06.89	07.07	07.36	06.91	07.10
iir_1_1	00.67	00.69	00.72	00.66	00.73
iir_4_64	03.85	04.08	04.47	03.84	03.98
insertsort	02.11	02.11	02.11	02.36	02.49
janne_complex	01.35	01.32	01.43	02.50	02.18
jfdctint	40:13.06	40:19.77	40:11.97	40:27.73	40:19.54
latnrm_32_64	49.02	01:41.55	38.96	01:34.97	01:37.92
latnrm_8_1	03.19	03.22	03.18	03.30	03.32
lms	15.04	16.44	20.61	17.01	19.52
lmsfir_32_64	04.81	04.88	05.43	04.84	05.06
lmsfir_8_1	02.50	02.57	02.83	03.16	02.81
lpc	05:43.77	05:11.97	05:46.71	08:54.29	12:08.79
ludcmp	04:20.83	05:18.69	03:51.80	05:39.66	01:28.87
matmult	05.51	06.19	09.92	12.03	35.00
minver	05:56.83	07:38.74	14:27.08	37:54.02	40:34.90
mult_10_10	02.48	02.58	02.53	02.49	02.75
mult_4_4	02.47	02.49	02.72	02.54	02.60
ndes	26.35	26.47	25.23	25.05	25.54
prime	01.85	01.88	02.00	01.93	01.89
qmf_receive	15.84	16.12	01:14.99	15.52	19.41
qmf_transmit	16.59	15.69	23.28	22.12	27.51
qsort-exam	40.47	02:36.17	42.51	07:27.25	10:19.37
qurt	05.76	06.13	05.75	06.41	06.86
searchmultiarray	02.33	02.40	02.34	02.45	02.75
select	08:30.14	02:53.21	17:01.59	40:18.15	40:22.04
selection_sort	02.49	02.61	02.52	03.49	03.28
sha	49.66	55.15	49.31	01:08.08	01:23.80
sqrt	02.89	02.97	03.08	02.88	02.98
st	13.69	14.21	14.07	14.21	14.77
v32.modem_achop	00.99	00.96	00.97	01.00	01.15
v32.modem_cnoise	04.22	04.23	04.21	04.16	04.56
v32.modem_eglue	00.81	00.81	00.82	00.86	00.91
Durchschnitt	03:56.94	04:04.17	04:02.79	06:01.14	06:05.12

K Messwerte 8

Tabelle K.1: ACET-Werte der Messreihen I und II [Zyklen]

Benchmark	RAGC	RAILP-CS	RAGC-WCET	RAILP-WCET
binarysearch	297	291	297	303
bsort100	360.459	350.560	360.459	356.097
cjpeg_jpeg6b_wrbmp	282.898	269.088	290.073	273.434
compressdata	3.013	2.889	2.817	3.131
countnegative	31.743	31.829	31.510	31.247
cover	31.392	30.533	31.392	30.146
crc	5.019	4.611	4.521	5.172
edn	73.267	73.510	66.650	68.789
fdct	8.361	3.630	8.374	3.581
fibcall	640	640	640	612
fir	8.092	6.217	5.210	6.821
fir_256_64	11.326	12.345	11.599	12.086
fir_32_1	765	763	739	769
hamming_window	44.213	33.701	32.715	34.084
iir_1_1	145	153	152	146
iir_4_64	59.148	33.856	39.835	34.493
insertsort	3.697	3.689	3.594	3.689
janne_complex	783	783	577	587
jfdctint	12.305	5.811	12.294	5.734
latnrm_32_64	280.065	187.001	237.529	182.653
latnrm_8_1	920	771	838	790
lms	834.065	765.639	838.509	798.686
lmsfir_8_1	550	544	573	486
lpc	540.944	473.989	390.360	477.199
ludcmp	8.298	6.498	6.498	6.437
matmult	480.349	418.150	410.874	420.144
mult_10_10	51.248	28.980	32.457	29.836
mult_4_4	2.477	1.731	2.178	1.792
ndes	145.237	134.122	123.085	134.122
prime	22.898	14.768	22.881	14.321
qurt	2.362	2.321	2.202	2.217
searchmultiarray	26.516	25.401	26.516	26.848
select	2.964	2.717	3.040	2.745
sqrt	8.150	8.418	7.833	8.373
st	370.799	374.627	367.775	370.764
v32.modem_achop	19.514	19.274	19.465	18.794
v32.modem_cnoise	134.923	129.651	139.550	132.010
v32.modem_eglue	29.431	27.280	29.431	27.041

Tabelle K.2: ACET-Werte der Messreihen VI, VII und VIII [Zyklen]

Benchmark	RAILP-WCET aus Messreihe VI	Kombination 1 aus Messreihe VII Verhältnis 90:10	Kombination 2 aus Messreihe VIII Verhältnis 80:20
binarysearch	308	299	296
bsort100	355.508	356.096	351.146
cjpeg_jpeg6b_wrbmp	274.712	267.786	268.040
compressdata	2.953	2.986	2.921
countnegative	31.691	31.026	30.547
cover	30.939	30.318	30.391
crc	5.297	5.098	5.106
edn	74.747	74.269	66.440
fdct	3.582	3.504	3.564
fibcall	640	640	640
fir	6.287	5.087	6.802
fir_256_64	12.079	11.574	11.829
fir_32_1	713	770	738
hamming_window	33.963	33.631	33.546
iir_1_1	145	146	145
iir_4_64	35.037	33.399	33.656
insertsort	3.732	3.742	3.643
janne_complex	645	670	670
jfdctint	5.802	5.758	5.703
latnrm_32_64	190.587	186.359	185.338
latnrm_8_1	781	809	806
lms	815.711	790.174	729.410
lmsfir_8_1	534	531	527
lpc	496.298	464.687	470.493
ludcmp	6.504	6.362	6.314
matmult	415.100	421.918	412.978
mult_10_10	32.814	31.504	32.822
mult_4_4	1.748	1.780	1.670
ndes	141.866	132.019	142.126
prime	17.755	14.747	18.595
qurt	2.563	2.230	2.227
searchmultiarray	27.473	29.696	29.698
select	2.870	2.679	2.815
sqrt	7.853	9.099	8.289
st	379.539	367.530	367.727
v32.modem_achop	19.223	19.222	19.463
v32.modem_cnoise	137.901	129.929	130.041
v32.modem_eglue	26.802	29.431	29.431

L Messwerte 9

Tabelle L.1: Anzahl der Instruktionen und Nebenbedingungen

Benchmark	Anzahl Instruktionen	Anzahl der Nebenbedingungen des RAILP-WCET aus Messreihe II	aus Messreihe VI	aus Messreihe VIII
adpcm_decoder	838	56.711	151.462	151.319
adpcm_encoder	884	60.433	162.182	162.012
binarysearch	43	3.868	7.802	7.782
bsort100	66	5.890	16.074	16.036
cjpeg_jpeg6b_transupp	733	138.732	159.455	422.118
cjpeg_jpeg6b_wrbmp	329	24.467	58.038	57.726
codecs_dcodhuff	425	29.866	81.516	81.265
compressdata	153	10.238	24.884	24.828
countnegative	113	11.510	28.653	28.609
cover	809	70.968	154.771	153.539
crc	185	14.629	42.450	42.392
edn	962	70.935	165.203	165.109
expint	89	8.498	27.964	27.925
fdct	201	19.200	61.811	61.797
fibcall	18	1.452	5.435	5.422
fir	64	7.492	18.530	18.502
fir_256_64	46	4.035	10.544	10.524
fir_32_1	39	3.134	6.714	6.703
g721.marcuslee_decoder	49	6.113	17.890	17.853
g721.marcuslee_encoder	63	8.668	25.820	25.769
h263	358	37.562	107.215	107.061
h264dec_ldecode_block	1.332	134.675	335.048	334.711
hamming_window	53	4.892	13.238	13.205
histogram	142	12.640	30.030	30.007
iir_1_1	39	2.358	4.379	4.373
iir_4_64	114	10.261	26.622	26.606
insertsort	63	4.589	11.266	11.253
janne_complex	33	2.747	9.076	9.039
jfdctint	221	21.374	68.170	68.155
latnrm_32_64	89	8.672	21.652	21.634
latnrm_8_1	73	6.120	15.285	15.271
lms	461	34.168	90.097	89.983
lmsfir_32_64	104	11.086	27.683	27.638
lmsfir_8_1	71	6.158	13.071	13.054
lpc	962	81.864	207.600	207.305
ludcmp	284	28.220	75.433	75.320
matmult	112	10.398	26.885	26.853
minver	331	37.770	106.819	106.640
mult_10_10	53	4.780	12.748	12.730
mult_4_4	55	4.892	11.584	11.572
ndes	639	49.753	122.967	122.804
prime	60	3.383	11.136	11.112
qmf_receive	136	17.058	43.573	43.525
qmf_transmit	139	17.587	45.251	45.202
qsort-exam	248	23.277	60.288	60.187
qurt	191	13.242	35.347	35.279
searchmultiarray	57	5.533	16.410	16.377
select	255	25.045	70.220	70.128
selection_sort	48	4.919	13.575	13.534
sha	806	72.450	193.898	193.658
sqrt	73	6.892	18.598	18.564
st	456	36.999	86.580	86.459
v32.modem_achop	47	2.929	7.877	7.841
v32.modem_cnoise	137	9.840	22.804	22.779
v32.modem_eglue	38	2.392	6.216	6.197

Abkürzungsverzeichnis

Abkürzung	Erwähnung auf Seite	ausgeschriebene Bezeichnung
ACET	14	Average Case Execution Time
ALU	37	Arithmetic Logic Unit
BB	10	Basisblock
BCET	14	Best-Case Execution Time
$BCET_{est}$	14	Estimated Best-Case Execution Time
$BCET_{obs}$	14	Observed Best-Case Execution Time
CFG	9	Control Flow Graph
CISC	21	Complex Instruction Set Computer
CPU	35	Central Processing Unit
CRL2	32	Control Flow Representation Language Version 2
DSP	35	Digitaler Signalprozessor
DLMB	36	Data Local Memory Bus
DMI	36	Data Memory Interface
DMU	35	Data Memory Unit
DP	38	Dual Pipeline
FP	37	Floating Point
FPU	35	Floating Point Unit
GPR	39	General Purpose Register
ICD	28	Informatik Centrum Dortmund
ICD-LLIR	30	ICD Low Level Intermediate Representation
IG	48	Instruction Graph
ILP	12	Integer Linear Programming
IP	37	Integer Pipeline
IPET	15	Implicit Path Enumeration Technique
LD	38	Load
LIR	28	Low Level Intermediate Representation
LS	38	Load/Store
LTA	10	Live Time Analysis
MAC	36	Multiply Accumulate
MG	54	Memory Graph
MIR	28	Medium Level Intermediate Representation
ORA	47	Optimal Register Allocation

Abkürzung	Erwähnung auf Seite	ausgeschriebene Bezeichnung
PBQP	5	Partitioned Boolean Quadratic Problem
PC	39	Program Counter
PLMB	35	Program Local Memory Bus
PMI	35	Program Memory Interface
PMU	35	Program Memory Unit
RAGC	30	Graph Coloring Register Allocation
RASA	31	Spilling All Register Allocation
RAGC-WCET	31	WCET-aware Graph Coloring Register Allocation
RAILP-WCET	31	WCET-aware ILP-based Register Allocation
RAILP-CS	31	Codesize-aware ILP-based Register Allocation
RAW	43	Read-After-Write
RISC	21	Reduced Instruction Set Computer
SPB	35	System Peripheral Bus
SPM	35	Scratchpad Memory
ST	38	Store
VRG	48	Virtual Register Graph
WAR	43	Write-After-Read
WAW	43	Write-After-Write
WCC	27	WCET-aware C-Compiler
WCEP	15	Worst-Case Execution Path
WCET	13	Worst-Case Execution Time
$WCET_{est}$	14	Estimated Worst-Case Execution Time
$WCET_{obs}$	14	Observed Worst-Case Execution Time

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Abs10] AbsInt Angewandte Informatik GmbH (Hrsg.): *a^3 AbsInt Advanced Analyzer for TriCore - User Documentation*. V10.04. Saarbrücken: AbsInt Angewandte Informatik GmbH, April 2010

[AG01] Appel, Andrew W. ; George, Lal: Optimal spilling for CISC machines with few registers. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 36 (2001), Nr. 5, S. 243–253

[App04] Appel, Andrew W.: *Modern compiler implementation in C*. 1. Paperback Edition. Cambridge University Press, 2004. – ISBN 0–521–60765–5

[ARM05] ARM Limited (Hrsg.): *ARM Architecture Reference Manual*. Ausgabe I. : ARM Limited, Juli 2005

[BCT92] Briggs, Preston ; Cooper, Keith D. ; Torczon, Linda: Rematerialization. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* 27 (1992), Nr. 7, S. 311–321

[BCT94] Briggs, Preston ; Cooper, Keith D. ; Torczon, Linda: Improvements to graph coloring register allocation. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 16 (1994), Nr. 3, S. 428–455

[BGS94] Bacon, David F. ; Graham, Susan L. ; Sharp, Oliver J.: Compiler transformations for high-performance computing. In: *ACM Computing Surveys (CSUR)* 26 (1994), Nr. 4, S. 345–420

[Bis09] Bisschop, Johannes: *AIMMS - Optimization Modeling*. 3. Ausgabe. Haarlem/Niederlande : Paragon Decision Technology B.V., 2009. – ISBN 978–1–84753–912–0

[CH90] Chow, Fred C. ; Hennessy, John L.: The priority-based coloring approach to register allocation. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12 (1990), Nr. 4, S. 501–536

[Cha81] Chaitin, G. J.: Register allocation via coloring. In: *Computer Languages* (1981), Nr. 6, S. 47–57

[Cor08] Cordes, Daniel: *Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib*, Technische Universität Dortmund, Diplomarbeit, April 2008. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/cordes.pdf. – Stand 19. Mai 2010

[Duf88] Duff, Tom: *Duff's Device*. http://www.lysator.liu.se/c/duffs-device.html. Version: August 1988. – Stand 31. Mai 2010

[EG06] Ermedahl, Andreas ; Gustafsson, Jan: *MRTC Benchmarks*. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html. Version: September 2006. – Stand 25. April 2010

[Fal09] Falk, Heiko: WCET-aware Register Allocation based on Graph Coloring. In: *The 46th Design Automation Conference (DAC)*, 2009, S. 726–731

[FK09] Falk, Heiko ; Kleinsorge, Jan C.: Optimal Static WCET-aware Scratchpad Allocation of Program Code. In: *The 46th Design Automation Conference (DAC)*, 2009, S. 732–737

[FL98] Farach, Martin ; Liberatore, Vincenzo: On local register allocation. In: *9th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1998, S. 564–573

[FLT06] Falk, Heiko ; Lokuciejewski, Paul ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, 2006, S. 121–126

[FPT07] Falk, Heiko ; Plazar, Sascha ; Theiling, Henrik: Compile Time Decided Instruction Cache Locking Using Worst-Case Execution Paths. In: *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, S. 143–148

[Fu07] Fu, Changqing: *A hybrid fast optimal register allocator*. Davis/USA, University of California at Davis, Diss., 2007

[FW02] Fu, Changqing ; Wilken, Kent D.: A faster optimal register allocator. In: *35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 2002, S. 245–256

[Ged08] Gedikli, Fatih: *Transformation und Ausnutzung von WCET-Informationen für High-Level-Optimierungen*, Technische Universität Dortmund, Diplomarbeit, September 2008. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/gedikli.pdf. – Stand 19. Mai 2010

[GRE+01] Guthaus, M. R. ; Ringenberg, J. S. ; Ernst, D. ; Austin, T. M. ; Mudge, T. ; Brown, R. B.: MiBench: A free, commercially representative embedded benchmark suite. In: *IEEE International Workshop on Workload Characterization (WWC)*, 2001, S. 3–14

[GW96] Goodwin, David W. ; Wilken, Kent D.: Optimal and near-optimal global register allocations using 0–1 integer programming. In: *Software - Practice and Experience (SPE)* 26 (1996), Nr. 8, S. 929–965

[HP03] Hennessy, John L. ; Patterson, David A.: *Computer Architecture. A Quantitative Approach*. 3. Edition. Morgan Kaufmann Publishers, 2003. – ISBN 978–1–55860–596–1

[HP07] Hennessy, John L. ; Patterson, David A.: *Computer Architecture. A Quantitative Approach*. 4. Edition. Morgan Kaufmann Publishers, 2007. – ISBN 978–0–12–370490–0

[HS06] Hames, Lang ; Scholz, Bernhard: Nearly optimal register allocation with PBQP. In: *7th Joint Modular Languages Conference (JMLC)*, 2006, S. 346–361

[ICD10a] ICD – Informatik Centrum Dortmund (Hrsg.): *ICD-C Compiler Framework Developer Manual*. Dortmund: ICD – Informatik Centrum Dortmund, April 2010

[ICD10b] ICD – Informatik Centrum Dortmund (Hrsg.): *ICD Low Level Intermediate Representation backend infrastructure (LLIR) Developer Manual*. Dortmund: ICD – Informatik Centrum Dortmund, April 2010

[Inf02] Infineon Technologies AG (Hrsg.): *TriCore™ 1 Architecture Overview Handbook*. V1.3.3. : Infineon Technologies AG, Mai 2002. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837. – Stand 22. April 2010

[Inf03a] Infineon Technologies AG (Hrsg.): *TriCore™ 1 DSP Optimization Guide - Part 1 Instruction Set*. V1.6.4. : Infineon Technologies AG, Januar 2003. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837. – Stand 22. April 2010

[Inf03b] Infineon Technologies AG (Hrsg.): *TriCore™ Compiler Writer's Guide*. V1.4. : Infineon Technologies AG, Dezember 2003. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837. – Stand 22. April 2010

[Inf04] Infineon Technologies AG (Hrsg.): *TriCore™ 1 Pipeline Behaviour & Instruction Execution Timing*. V1.1. : Infineon Technologies AG, Juni 2004. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837. – Stand 22. April 2010

[Inf07] Infineon Technologies AG (Hrsg.): *TC1796 User's Manual System and Peripheral Units*. V2.0. : Infineon Technologies AG, Juli 2007. http://www.infineon.com/cms/de/product/channel.html?channel=ff80808112ab681d0112ab6b6aaf0819&tab=2. – Stand 22. April 2010

[Inf08] Infineon Technologies AG (Hrsg.): *TriCore™ 1 Architecture - Volume 1 Core Architecture*. V1.3.8. : Infineon Technologies AG, Januar 2008. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837. – Stand 22. April 2010

[Inf09] Infineon Technologies AG (Hrsg.): *TC1797 User's Manual*. V1.1. : Infineon Technologies AG, Mai 2009. http://www.infineon.com/cms/en/product/channel.html?channel=db3a3043156fd5730115b88e83410e81&tab=2. – Stand 02. Mai 2010

[ISO99] ISO/IEC (Hrsg.): *International Standard 9899 'Programming languages - C'*. Second Edition. : ISO/IEC, Dezember 1999

[Kal02] Kallrath, Josef: *Gemischt-ganzzahlige Optimierung*. 1. Ausgabe. Braunschweig : Vieweg Verlag, 2002. – ISBN 3–528–03141–7

[Kel09] Kelter, Timon: *Superblock-basierte High-Level WCET-Optimierungen*, Technische Universität Dortmund, Diplomarbeit, September 2009. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/kelter.pdf. – Stand 18. Mai 2010

[Kle08] Kleinsorge, Jan C.: *WCET-centric code allocation for scratchpad memories*, Technische Universität Dortmund, Diplomarbeit, September 2008. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/kleinsorge.pdf. – Stand 22. April 2010

[Kop00] Kopetz, Hermann: Software engineering for real-time: a roadmap. In: *Conference on The Future of Software Engineering (ICSE)*, 2000, S. 201–211

[KP05] Kirner, Raimund ; Puschner, Peter: Classification of WCET Analysis Techniques. In: *8th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2005, S. 190–199

[KW98] Kong, Timothy ; Wilken, Kent D.: Precise register allocation for irregular architectures. In: *31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, 1998, S. 297–307

[Lee98] Lee, Corinna G.: *UTDSP Benchmark Suite*. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html. Version: Mai 1998. – Stand 25. April 2010

[LFM08] Lokuciejewski, Paul ; Falk, Heiko ; Marwedel, Peter: WCET-driven Cache-based Procedure Positioning Optimizations. In: *The 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008, S. 321–330

[LFMT08] Lokuciejewski, Paul ; Falk, Heiko ; Marwedel, Peter ; Theiling, Henrik: WCET-Driven, Code-Size Critical Procedure Cloning. In: *The 11th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, 2008, S. 21–30

[LG97] Lueh, Guei-Yuan ; Gross, Thomas: Call-cost directed register allocation. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 1997, S. 296–307

[LGM09] Lokuciejewski, Paul ; Gedikli, Fatih ; Marwedel, Peter: Accelerating WCET-driven Optimizations by the Invariant Path - a Case Study of Loop Unswitching. In: *The 12th International Workshop on Software & Compilers for Embedded Systems (SCOPES)*, 2009, S. 11–20

[LGMM09] Lokuciejewski, Paul ; Gedikli, Fatih ; Marwedel, Peter ; Morik, Katharina: Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining. In: *3rd Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, 2009, S. 1–15

[LGZ+09] Lv, Mingsong ; Guan, Nan ; Zhang, Yi ; Deng, Qingxu ; Yu, Ge ; Zhang, Jianming: A Survey of WCET Analysis of Real-Time Operating Systems. In: *International Conference on Embedded Software and Systems (ICESS)*, 2009, S. 65–72

[LM95] Li, Yau-Tsun S. ; Malik, Sharad: Performance analysis of embedded software using implicit path enumeration. In: *ACM SIGPLAN Workshop on Languages, Compilers, & Tools for Embedded Systems (LCTES)*, 1995, S. 88–98

[LM09] Lokuciejewski, Paul ; Marwedel, Peter: Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. In: *The 21st Euromicro Conference on Real-Time Systems (ECRTS)*, 2009, S. 35–44

[LPF+10] Lokuciejewski, Paul ; Plazar, Sascha ; Falk, Heiko ; Marwedel, Peter ; Thiele, Lothar: Multi-Objective Exploration of Compiler Optimizations

for Real-Time Systems. In: *Proceedings of the 13th International Symposium on Object/Component/Service-oriented Real-time Distributed Computing (ISORC)*, 2010, S. 115–122

[LPMS97] Lee, Chunho ; Potkonjak, Miodrag ; Mangione-Smith, William H.: MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In: *30th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 30)*, 1997, S. 330–335

[LSMM10] Lokuciejewski, Paul ; Stolpe, Marco ; Morik, Katharina ; Marwedel, Peter: Automatic Selection of Machine Learning Models for WCET-aware Compiler Heuristic Generation. In: *Proceedings of the 4th Workshop on Statistical and Machine Learning Approaches to Architectures and Compilation (SMART)*, 2010, S. 3–17

[Mar08] Marwedel, Peter: *Eingebettete Systeme*. korrigierter Nachdruck. Berlin : Springer Verlag, 2008. – ISBN 978–3–540–34048–5

[Mar09] Marwedel, Peter: *Foliensatz der Vorlesung „Eingebettete Systeme“*. http://ls12-www.cs.tu-dortmund.de/staff/marwedel/es-book/slides09/es-marw-1.1.pdf. Version: Oktober 2009. – Stand 19. Juni 2010

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design & Implementation*. 1. Auflage. San Diego/USA : Morgan Kaufmann Publishers, 1997. – ISBN 1–55860–320–4

[Nin93] Ning, Qi: Register allocation for optimal loop scheduling. In: *Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 1993, S. 942–955

[PLM09] Plazar, Sascha ; Lokuciejewski, Paul ; Marwedel, Peter: WCET-aware Software Based Cache Partitioning for Multi-Task Real-Time Systems. In: *The 9th International Workshop on Worst-Case Execution Time Analysis (WCET)*, 2009, S. 78–88

[PP07] Puaut, Isabelle ; Pais, Christophe: Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In: *Conference on Design, Automation and Test in Europe (DATE)*, 2007, S. 1484–1489

[PS99] Poletto, Massimiliano ; Sarkar, Vivek: Linear scan register allocation. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21 (1999), Nr. 5, S. 895–913

[QPP08] Quintão Pereira, Fernando M. ; Palsberg, Jens: Register allocation by puzzle solving. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2008, S. 216–226

[Rot08] Rotthowe, Felix: *Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung*, Technische Universität Dortmund, Diplomarbeit, August 2008. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/rotthowe.pdf. – Stand 19. Mai 2010

[SB07] Sarkar, Vivek ; Barik, Rajkishore: Extended linear scan: an alternate foundation for global register allocation. In: *16th International Conference on Compiler Construction (CC)*, 2007, S. 141–155

[Sch07] Schulte, Daniel: *Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler*, Technische Universität Dortmund, Diplomarbeit, Mai 2007. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/schulte.pdf. – Stand 16. Mai 2010

[Sch08] Schmoll, Florian: *ILP-basierte Registerallokation unter Ausnutzung von WCET-Daten*, Technische Universität Dortmund, Diplomarbeit, September 2008. http://ls12-www.cs.tu-dortmund.de/publications/theses/downloads/schmoll.pdf. – Stand 22. April 2010

[SE02] Scholz, Bernhard ; Eckstein, Erik: Register allocation for irregular architectures. In: *Joint Conference on Languages, Compilers and Tools for Embedded Eystems (LCTES/SCOPES)*, 2002, S. 139–148

[Set73] Sethi, Ravi: Complete register allocation problems. In: *5th Annual ACM Symposium on Theory of Computing (STOC)*, 1973, S. 182–195

[SMRC05] Suhendra, Vivy ; Mitra, Tulika ; Roychoudhury, Abhik ; Chen, Ting: WCET Centric Data Allocation to Scratchpad Memory. In: *26th IEEE International Real-Time Systems Symposium (RTSS)*, 2005, S. 223–232

[THS98] Traub, Omri ; Holloway, Glenn ; Smith, Michael D.: Quality and speed in linear-scan register allocation. In: *ACM Special Interest Group on Programming Languages (SIGPLAN)* 33 (1998), Nr. 5, S. 142–151

[WEE+08] Wilhelm, Reinhard ; Engblom, Jakob ; Ermedahl, Andreas ; Holsti, Niklas ; Thesing, Stephan ; Whalley, David ; Bernat, Guillem ; Ferdinand, Christian ; Heckmann, Reinhold ; Mitra, Tulika ; Mueller, Frank ; Puaut, Isabelle ; Puschner, Peter ; Staschulat, Jan ; Stenström, Per: The Worst-Case Execution-Time Problem—Overview of Methods and Survey of Tools. In: *ACM Transactions on Embedded Computing Systems (TECS)* 7 (2008), Nr. 3, S. 1–53

[Weg96] Wegener, Ingo: *Kompendium Theoretische Informatik - eine Ideensammlung*. 1. Ausgabe. Stuttgart : B. G. Teubner, 1996. – ISBN 3–519–02145–5

[Weg03] Wegener, Ingo: *Komplexitätstheorie*. 1. Ausgabe. Berlin : Springer Verlag, 2003. – ISBN 3–540–00161–1

[WF10] Wimmer, Christian ; Franz, Michael: Linear scan register allocation on SSA form. In: *8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2010, S. 170–179

[WM05] Wimmer, Christian ; Mössenböck, Hanspeter: Optimized interval splitting in a linear scan register allocator. In: *1st ACM/USENIX international conference on Virtual execution environments (VEE)*, 2005, S. 132–141

[ZHC02] Zhang, Yumin ; Hu, Xiaobo ; Chen, Danny Z.: Efficient global register allocation for minimizing energy consumption. In: *ACM SIGPLAN Notices* 37 (2002), Nr. 4, S. 42–53

[ZKW+06] Zhao, Wankang ; Kreahling, William ; Whalley, David ; Healy, Christopher ; Mueller, Frank: Improving WCET by applying worst-case path optimizations. In: *Real-Time Systems* 34 (2006), Nr. 2, S. 129–152