TECHNISCHE UNIVERSITÄT DORTMUND
FAKULTÄT FÜR INFORMATIK

Diplomarbeit

Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung

Felix Rotthowe

11. August 2008

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik 12
Technische Informatik und Eingebettete Systeme
Fakultät für Informatik
Technische Universität Dortmund

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

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

Kapitel 1

Einführung

Der Einfluss von Informationstechnologie auf unser tägliches Leben ist seit Jahren stetig gestiegen und ein Stoppen dieser Bewegung ist nicht abzusehen. Inzwischen nutzen über 70 Prozent der deutschen Bevölkerung regelmäßig einen Computer [Sta07]. Noch viel höher ist der Anteil der Personen, die täglich unbewusst mit Rechnern in Berührung kommen, sei es in ihrem Mobiltelefon, Automobil oder sogar der Kaffeemaschine. Hierbei handelt es sich um so genannte **eingebettete Systeme**, das sind informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind [Mar07].

Bis auf wenige Ausnahmen muss bei der Entwicklung von eingebetteten Systemen sowohl Hardware, als auch die auf dieser Hardware laufende Software entworfen werden. Für beide Aufgabenbereiche existieren zahlreiche Tools, die den Entwickler beim Entwurfsvorgang unterstützen und selbstständig Optimierungen vornehmen können. Die in dieser Diplomarbeit behandelte Optimierung ist in einen Compiler, ein Tool zur Softwareentwicklung, integriert. Dennoch können die erzielten Ergebnisse eine Hilfestellung bei der Auswahl der benötigten Hardware bieten.

1.1 Motivation

Viele Bereiche, in denen eingebettete Systeme eingesetzt werden, erfüllen sicherheitskritische Aufgaben. Der Airbag-Controller eines Autos oder die Steuerung medizinischer Laser sind zwei Beispiele für den vielfältigen Aufgabenbereich, in dem Maschinen das Leben eines Menschen retten, aber auch gefährden können. Daher müssen diese Systeme in höchstem Maße verlässlich sein. Dazu gesellen sich unzählige Anwendungen, in denen der Ausfall eines Systems oder aber auch nur eine verspätete Reaktionszeit für den Benutzer unangenehm ist und somit vermieden werden muss. Ein Mobiltelefon beispielsweise sollte die eingehenden Sprachdaten in Echtzeit dekodieren können, um Aussetzer zu verhindern. Auch die Positionsangaben eines GPS-Empfängers sollten in einem Navigationssystem rechtzeitig zur Verfügung stehen, um eine brauchbare Zielführung zu gewährleisten.

Es ist offensichtlich, dass bei der Entwicklung von eingebetteten Systemen solche Echtzeit-Bedingungen berücksichtigt werden müssen, um ein sicheres und kommerziell erfolgreiches Produkt zu schaffen. Dies steht jedoch zuweilen im Widerspruch zu anderen Anforderungen. In vielen Fällen werden eingebettete Systeme mobil und damit ohne Anschluss an das Stromnetz genutzt. Da die Energie dann über Batterien bezogen wird, muss diese beschränkte Ressour-

ce äußerst effizient eingesetzt werden. Zusätzlich soll das fertige Produkt natürlich möglichst kostengünstig sein. Aus diesen Punkten ergibt sich, dass eine grobe Überdimensionierung der Hardware nicht der Weg sein kann, der zur Zusicherung der benötigten Reaktionszeit gewählt werden sollte. Vielmehr sollte die benötigte Rechenzeit der Software zur Entwicklungszeit bekannt sein und gezielt minimiert werden. Dann kann eine angemessene Hardware eingesetzt werden, um Kosten- und Energieaufwand minimal zu halten.

In diesem Zusammenhang spielt die durchschnittliche Ausführungszeit eines Programms eine eher untergeordnete Rolle. Wichtig ist vor allem, dass ein System auch im ungünstigsten Fall zuverlässig arbeitet und alle Zeitschranken eingehalten werden. Ziel ist es also, die maximale Programm-Laufzeit (*worst-case execution time, WCET*) zu minimieren. Ist die Hardware eines Systems gegeben, kann dies nur durch Optimierung des Programmcodes geschehen. Dies kann einerseits der Entwickler selbst erledigen, indem er zeitkritische Stellen analysiert und entsprechend verbessert. Dazu stehen eine Reihe von Tools zur Verfügung, die den kritischen Ausführungspfad berechnen und so wichtige Anhaltspunkte darüber liefern, an welcher Stelle eine Optimierung sinnvoll ist. Aufgrund der steigenden Komplexität eingebetteter Systeme stößt dieses Vorgehen jedoch bald an seine Grenzen. Während die Software eines durchschnittlichen Automobils in den 70er Jahren noch aus ca. 100.000 Zeilen Programmcode bestand, sind es heutzutage schon über eine Million; bis 2010 soll diese Zahl sogar auf 100 Millionen wachsen [Kel07]. Um diesem Anstieg an Komplexität Herr zu werden, ohne schlecht wartbaren und damit fehleranfälligen Code zu erzeugen, muss die Softwareentwicklung auf immer höheren Abstraktionsebenen stattfinden. So ist im Gegensatz zu früher der Einsatz von Hochsprachen auch für maschinennahe Anwendungen üblich. Zusätzlich erfreuen sich graphische Entwicklungstools immer größerer Beliebtheit. Ein Beispiel ist die Software *Statemate* [Sta08]. Hier modelliert der Entwickler ein System als Zustandsübergangsdiagramm. Der Programmcode wird dann aus dieser Beschreibung automatisch erzeugt.

Durch diese Verlagerung auf höhere Abstraktionsebenen verliert der Programmierer jedoch immer mehr den Einfluss auf den Maschinencode, der tatsächlich später auf der Zielplattform läuft. Daher ist die automatische WCET-Minimierung durch den Compiler ein sehr lohnenswertes Ziel und wird in Zukunft noch weiter an Bedeutung gewinnen.

Ein weiteres Problem, das sich durch den Bedarf nach immer schnelleren und leistungsfähigeren Systemen ergibt, ist die zunehmende Geschwindigkeitsdifferenz zwischen Prozessor und Hauptspeicher. Um diese auszugleichen, werden häufig Speicherhierarchien eingesetzt, in denen zusätzlich zu dem großen, aber langsamen Hauptspeicher kleinere und schnellere Speicher vorhanden sind. Hierzu zählen Caches und Scratchpad-Speicher. Während Caches zur Laufzeit eines Programms Code oder Daten selbstständig ein- und auslagern, ist die Nutzung von Scratchpad-Speichern im Allgemeinen dem Entwickler überlassen. Dieser zusätzliche Aufwand ist dennoch lohnenswert, denn Scratchpad-Speicher haben nicht nur einen geringeren Energieverbrauch als Caches, sondern erhöhen außerdem die Vorhersagbarkeit eines Programms, was eine genauere WCET-Analyse ermöglicht [WM05].

Eine vom Compiler automatisch durchgeführte Scratchpad-Allokation befreit den Entwickler von dieser Aufgabe und vereint somit die Vorteile von Caches und Scratchpad-Speichern. Durch gezielte Auswahl der auszulagernden Objekte kann zudem die maximale Ausführungszeit des Programms minimiert und so die Kosten des Systems möglichst gering gehalten werden.

1.2 Verwandte Arbeiten

Die Verlagerung von Daten in Scratchpad-Speicher ist inzwischen ein relativ weit erforschtes Gebiet. Die meisten der vorhandenen Arbeiten beschäftigen sich allerdings mit der Optimierung der durchschnittlichen Programmausführungszeit (engl. Average-Case Execution Time, ACET). Panda et. al beschreiben in [PND98] Methoden, um den Scratchpad-Speicher eines Systems, das zusätzlich über einen DRAM mit vorgeschaltetem Datencache verfügt, effizient zu nutzen. Die Autoren von [ABS02] stellen einen ILP-Ansatz zur optimalen statischen Allokation von globalen und Stack-Variablen vor. In [UB03] wird eine Erweiterung dieses Ansatzes untersucht, die eine Änderung des Scratchpad-Inhaltes zur Laufzeit des Programms erlaubt (*dynamische Allokation*).

Steinke et al. minimieren den Energieverbrauch durch die Platzierung von Daten und Code im Scratchpad-Speicher [SWLM02]. In [VWM04] wird ein darauf aufbauendes Verfahren zur dynamischen Allokation vorgestellt. Es konnte gezeigt werden, dass die Ergebnisse dieser Optimierung auch positiven Einfluss auf die Bestimmung der WCET haben, da die Vorhersagbarkeit von Programmen im Gegensatz zur Verwendung von Caches deutlich gesteigert werden kann [WM04].

Die gezielte Optimierung der Worst-Case Execution Time ist ein erst vor vergleichsweise kurzer Zeit von der Forschung aufgegriffenes Thema. Einer der Gründe hierfür ist vermutlich, dass brauchbare Analyse-Tools zur Bestimmung der WCET von Programmen erst in den letzten Jahren verfügbar wurden [FH04, CP01, Erm03].

Das Thema dieser Arbeit ist die WCET-Minimierung durch Auslagerung von Daten in den Scratchpad-Speicher. Suhendra et al. untersuchen in [SMRC05] drei verschiedene Ansätze zur Berechnung einer statischen Scratchpad-Allokation. Neben einem Greedy-Verfahren stellen sie ein ILP-basiertes Verfahren und einen Branch-and-Bound Algorithmus vor. Sie zeigen, dass durch alle drei Ansätze große Verbesserungen der WCET im Vergleich zu einer ACET-basierten Allokation erreicht werden können. Die vorliegende Arbeit greift das von Suhendra et al. vorgestellte ILP-Verfahren auf und erweitert es in wesentlichen Punkten, so dass eine praktische Anwendung auf nahezu beliebige Programme und die nahtlose Integration in einen Compiler ermöglicht wird.

Eine dynamische Scratchpad-Allokation zur WCET-Optimierung stellen Deverge und Puaut vor [DP07]. Sie berechnen den Worst-Case Execution Pfad aus den Daten einer WCET-Analyse und stellen dann ein ILP auf, durch das eine günstige Scratchpad-Belegung für jede Kante des Kontrollflussgraphen berechnet wird. Die Instabilität des Worst-Case Execution Pfades behandeln sie durch eine iterative Vorgehensweise: Nach einer weiteren WCET-Analyse wird erneut ein ILP formuliert und so falls möglich zusätzliche auszulagernde Objekte bestimmt. Dieses Vorgehen wiederholen sie, bis kein Platz im Scratchpad-Speicher mehr vorhanden ist. Dies entspricht einem Greedy-Verfahren, da die entlang des ersten berechneten Worst-Case Pfades bestimmten Allokationen nicht mehr zurückgenommen werden können.

Dem Autor dieser Arbeit ist kein Projekt bekannt, das die vollständige Integration einer WCET-optimierenden Scratchpad-Allokation von Daten in einen Compiler realisiert. Möglich wird dies erst, wenn WCET-Informationen innerhalb des Compilers zur Verfügung stehen. Eine Realisierung dieser Aufgabe stellen Falk, Lokuciejewski und Theiling in [FLT06] vor.

Um einen Überblick über das relativ junge Forschungsgebiet der WCET-Optimierung zu geben,

sollen nun noch kurz einige weitere Forschungsarbeiten erwähnt werden, die sich mit diesem Thema beschäftigen, jedoch nicht direkt mit dieser Arbeit im Zusammenhang stehen. Lee et al. stellen in [LLPM04] eine Methode vor, welche auf einem *Dual Instruction Set* ARM Prozessor durch geschickte Instruktionsauswahl eine Minimierung der WCET ermöglicht und gleichzeitig die Codegröße gering hält. Die Größenreduzierung wird durch die Nutzung des platzsparenden *Thumb*-Befehlssatzes auf nicht kritischen Pfaden erreicht, während entlang des Worst-Case Pfades die Standardinstruktionen genutzt werden, um eine schnelle Ausführung zu gewährleisten. Zhao et al. optimieren die WCET durch *Code Positioning* [ZWHM05]: Die Speicherorte von Basisblöcken werden so umsortiert, dass möglichst viele Blöcke entlang des kritischen Pfades ohne Sprünge durchlaufen werden können. Eine WCET-Optimierung durch *Loop Nest Splitting* stellen Falk et al. vor [FS06].

1.3 Ziel der Diplomarbeit

Am Lehrstuhl 12 für Technische Informatik und Eingebettete Systeme der Technischen Universität Dortmund wurde der *WCET-aware C Compiler (WCC)* entwickelt, mit dem durch die Integration des WCET-Analyse Tools *aiT* gezielt WCET-Optimierungen entwickelt und untersucht werden können.

Ziel dieser Diplomarbeit ist die Implementierung eines Algorithmus zur Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung und die Integration dieser Optimierung in den WCC. Im Detail sollen folgende Anforderungen erfüllt werden:

- Es soll eine WCET-Optimierung durch Auslagerung von globalen und statischen Daten in den On-Chip Scratchpad-Speicher der Zielarchitektur des WCC erreicht werden. Die auszulagernden Datenobjekte sollen zur Übersetzungszeit bestimmt werden.
- Als Grundlage der Optimierung sind die durch die Integration des WCET-Analyse Tools *aiT* in den Compiler gewonnenen Informationen zu nutzen. Weiterhin werden die durch Programmannotationen spezifizierten und im WCC über spezielle Datenstrukturen bereitgestellten Kontrollflussinformationen (z.B. Schleifeniterationsgrenzen) genutzt.
- Zusätzlich werden Informationen darüber benötigt, auf welche Datenobjekte durch die einzelnen Instruktionen des Programms zugegriffen wird. Diese Daten müssen in geeigneter Form gewonnen werden und können dann für alle Komponenten des Compilers zur Verfügung gestellt werden. Dies ermöglicht auch die Erforschung weiterer datenbezogener Optimierungen im WCC im Anschluss an diese Arbeit.
- Die Optimierung soll über ein Kommandozeilen-Flag aktiviert werden können und ohne weitere Interaktion des Nutzers durchgeführt werden. Somit wird erstmals eine komplette Integration einer WCET-optimierenden Scratchpad-Allokation in einen Compiler erreicht.
- Die Ergebnisse der Optimierung sollen anhand von Benchmarks für verschiedene Scratchpad-Größen dokumentiert und bewertet werden.

Zunächst soll eine statische Allokation realisiert werden. Es ist jedoch wünschenswert, den Inhalt des Scratchpad-Speichers zur Laufzeit eines Programms ändern zu können, um eine effizientere Nutzung zu erlauben. Daher wurde das optionale Ziel gesetzt, ein Verfahren zur dynamischen Allokation zu entwickeln. Dies erfordert neben der Berücksichtigung des wechselbaren

Scratchpad-Inhalts bei der Bestimmung von auszulagernden Datenobjekten auch die technisch aufwändige Implementierung eines Verfahrens, diese Objekte zur Laufzeit zwischen den Speichern zu kopieren. Durch eine Erweiterung des vorgestellten ILP-Verfahrens wird erstmals eine WCET-Optimierung durch dynamische Scratchpad-Allokation von Daten vorgestellt, bei der eine gleichzeitige Berücksichtigung aller möglichen Kontrollflusspfade erfolgt, anstatt die Auswahl der auszulagernden Objekte lediglich entlang des aktuellen Worst-Case Execution Pfades zu treffen.

Schließlich runden der Vergleich von statischer und dynamischer Allokation und die Untersuchung deren praktischer Einsetzbarkeit die in dieser Arbeit gewonnenen Forschungsergebnisse ab.

1.4 Aufbau der Diplomarbeit

Der restliche Teil dieser Arbeit ist wie folgt strukturiert:

- **Kapitel 2** gibt einen Überblick über die theoretischen Grundlagen dieser Arbeit. Dazu wird zunächst in die Nutzung von Scratchpad-Speichern zum schnelleren Zugriff auf Daten eingeführt. Daraufhin wird die Metrik der Worst-case Execution Time behandelt und auf die Herausforderungen bei deren Bestimmung eingegangen. Schließlich sollen lineare Optimierungsprobleme, welche die Grundlage des in dieser Arbeit verwendeten Algorithmus darstellen, betrachtet werden.
- **Kapitel 3** stellt das Framework vor, in das die Optimierung integriert ist. Dazu gehört die Beschreibung der Zielplattform, die Vorstellung des verwendeten WCET-Analysetools aiT und die Beschreibung des Compilers WCC. Somit werden in diesem Kapitel die praktischen Grundlagen der vorliegenden Arbeit erläutert.
- **Kapitel 4** beschreibt den ILP-Algorithmus zur statischen Auslagerung von Datenobjekten in den Scratchpad-Speicher. Hier wird bewusst von plattformspezifischen Details abstrahiert, da das vorgestellte Verfahren in dieser Form sehr universell einsetzbar ist.
- **Kapitel 5** behandelt die praktische Umsetzung der Optimierung innerhalb des WCC. An dieser Stelle wird zunächst der grobe Aufbau des Algorithmus und dessen Integration in den Compiler beleuchtet. Im Folgenden werden dann die Methoden zur Gewinnung der Eingabedaten des Algorithmus vorgestellt und Details zu dessen Implementierung gegeben.
- In **Kapitel 6** wird die Entwicklung einer dynamischen Allokation beschrieben. Dazu wird eine Erweiterung des statischen Algorithmus vorgestellt, die eine Änderung des Scratchpad-Inhalts zur Laufzeit unter Berücksichtigung der anfallenden Kosten des Kopierens von Datenobjekten zwischen den Speichern ermöglicht. Im Anschluss daran findet sich die Beschreibung der Umsetzung dieses Verfahrens.
- **Kapitel 7** stellt die durch die Optimierungen erzielten Ergebnisse vor. Anhand von Benchmarks wird der Einfluss der Optimierungen auf die WCET für unterschiedliche Scratchpad-Größen gezeigt.
- **Kapitel 8** schließlich fasst die in dieser Arbeit gewonnen Erkenntnisse zusammen und gibt einen Ausblick auf mögliche darauf aufbauende Arbeiten.

Kapitel 2

Grundlagen

Dieses Kapitel führt in die theoretischen Grundlagen der vorliegenden Arbeit ein. Zunächst wird kurz auf die Verwendung von Scratchpad-Speichern in eingebetteten Systemen eingegangen. Im darauf folgenden Abschnitt wird die Metrik der Worst-Case Execution Time eingeführt und Methoden zu deren Berechnung betrachtet. Schließlich soll die im Rahmen dieser Arbeit verwendete Methode der ganzzahligen linearen Optimierung beschrieben werden.

2.1 Scratchpad Speicher

Damit die Leistungsfähigkeit eines Prozessors ausgereizt werden kann, sollten Code und Daten für diesen möglichst unverzüglich zugreifbar sein. Dieses Ziel ist in den meisten Fällen leider nicht erreichbar. Während die Geschwindigkeit von Prozessoren weiterhin rapide zunimmt, verläuft das Wachstum der Speichergeschwindigkeit deutlich langsamer und große Hauptspeicher können schon lange nicht mehr ohne Wartezyklen angesprochen werden [WM95]. Hinzu kommt, dass moderne Programme immer mehr Speicher benötigen, größere Speicher jedoch langsamer sind. Als Ausweg aus diesem Dilemma kann zusätzlich zum Hauptspeicher ein kleinerer, schnellerer Speicher eingesetzt werden, der häufig direkt auf dem CPU-Chip integriert ist und mit der Taktrate des Prozessors angesteuert werden kann. Der Inhalt dieses Speichers sollte natürlich genau die Daten enthalten, auf die besonders häufig zugegriffen wird. Dieses Verfahren kann auch mehrstufig angewendet werden, so dass man von Speicherhierarchien spricht.

In Desktoprechnern und Servern werden zu diesem Zweck fast ausschließlich Caches eingesetzt. Diese haben den Vorteil, dass sie transparent für den Entwickler agieren und selbstständig Entscheidungen zum Ein- und Auslagern von Daten aus dem Hauptspeicher treffen. Dies ist bei den oben genannten Rechnertypen nötig, da zur Übersetzungszeit beispielsweise der genaue Prozessortyp und die Hauptspeichergröße des Zielsystems nicht bekannt sind. Ein Programm kann somit mit Hilfe von Caches auf vielen verschiedenen Systemen eine hohe Performance erreichen.

Caches haben jedoch auch Nachteile. Durch die zusätzliche Logik benötigen sie viel Energie und das Programmverhalten kann oft nicht genau vorhergesagt werden. Im Embedded-Bereich, in dem neben der Energieeffizienz auch die Verlässlichkeit eine große Rolle spielt, wirken sich diese Nachteile deutlicher aus. Dennoch muss auch hier die Lücke zwischen Prozessor- und Hauptspeichergeschwindigkeit ausgeglichen werden. Zu diesem Zweck werden häufig Scratchpad-

Speicher verwendet, die in diesem Abschnitt behandelt werden.

Scratchpad-Speicher sind kleine Speicher, die in den Adressraum des Prozessors eingeblendet werden. Diese Speicher verfügen über keinerlei Logik zur Ein- oder Auslagerung von Objekten. Die Nutzung von Scratchpad-Speichern ist also voll und ganz dem Programm selbst überlassen. Dies bedeutet in der Regel einen Mehraufwand für den Entwickler, der selbst die Objekte zu identifizieren hat, deren Auslagerung den größten Gewinn bringt. Dies kann nicht nur mit dem Ziel der Minimierung der durchschnittlichen Programmlaufzeit, sondern mit beliebigen Absichten geschehen, beispielsweise der in dieser Arbeit beschriebenen Optimierung der WCET. Eine solche Optimierung kann natürlich auch durch den Compiler vorgenommen werden. Man unterscheidet hier zwischen statischer und dynamischer Allokation.

Eine statische Allokation bestimmt eine Teilmenge der Programmobjekte, die beim Start des Programms einmalig in den Scratchpad-Speicher geladen werden und dort über die gesamte Laufzeit verweilen. Die Anwendung dieses Verfahrens bringt keinen Overhead mit sich, da sich aus Sicht des Programmcodes nur die Speicheradressen von Daten oder auszuführendem Code ändern. Kann also auf ein Objekt im Hauptspeicher mit einer Latenzzeit von t_{dram} und auf ein Objekt im Scratchpad mit der Latenz t_{spm} zugegriffen werden, so beträgt der Laufzeitgewinn pro Zugriff $t_{dram} - t_{spm}$. Der zu erwartende Gewinn einer statischen Optimierung ist dann das Produkt dieses Gewinns mit der Anzahl der Zugriffe, die in den schnelleren Speicher erfolgen.

Bei einer dynamischen Allokation hingegen kann der Inhalt des Scratchpads auch zur Laufzeit des Programms geändert werden. Hierdurch ist eine effiziente Nutzung des schnelleren Speichers möglich, da immer die gerade vom Programm benötigten Daten vorgehalten werden können. Dies ist vergleichbar mit dem Verhalten eines Caches, bietet aber dennoch den Vorteil, dass der Speicherort der Objekte an jedem Programmpunkt zur Übersetzungszeit genau bekannt ist.

Um dieses Verfahren einzusetzen, müssen Programmpunkte ermittelt werden, an denen Objekte aus dem Hauptspeicher in den Scratchpad und zurück kopiert werden. Dies kann, je nach Ansatz, an den Grenzen von Funktionen, Schleifen oder sogar Instruktionen erfolgen. Für das Kopieren der Objekte zwischen den unterschiedlichen Speichern ist zusätzlicher Code, so genannter *Spillcode*, erforderlich. Da dieser Code selbst zur Laufzeit beiträgt, ist durch ein geeignetes Kostenmodell zu gewährleisten, dass der Gewinn durch Zugriffe auf den schnelleren Speicher die Kosten der Ein- und Auslagerung übertrifft.

2.2 WCET

Systeme, an die Echtzeit-Anforderungen gestellt werden, müssen in jedem Fall vor einer gegebenen Deadline reagieren. Insbesondere schließt dies ein, dass Berechnungen auch in unvorhergesehenen Situationen rechtzeitig terminieren. Ein zu spät geliefertes Ergebnis kann im schlimmsten Fall die gleichen Auswirkungen haben wie ein falsches Ergebnis. Daher ist bei der Entwicklung von Echtzeit-Systemen dafür Sorge zu tragen, dass auch im ungünstigsten Fall alle Zeitschranken eingehalten werden. Die Laufzeit eines Programms in diesem Fall nennt man Worst-Case Execution Time (WCET).

2.2.1 WCET Analyse

Das Ziel der WCET-Analyse ist die Berechnung einer möglichst genauen oberen Schranke der WCET eines Programms. Eine optimale Analyse würde für jedes Programm die exakte WCET berechnen. Dies ist im Allgemeinen nicht möglich, daher muss versucht werden, eine möglichst genaue Abschätzung zu berechnen, die aber dennoch sicher ist, also oberhalb des tatsächlichen Wertes liegt (siehe Abb. 2.1). Puschner und Burns geben in [PB00] einen Überblick über die Ziele und Probleme der Worst-Case Execution Time Analyse.

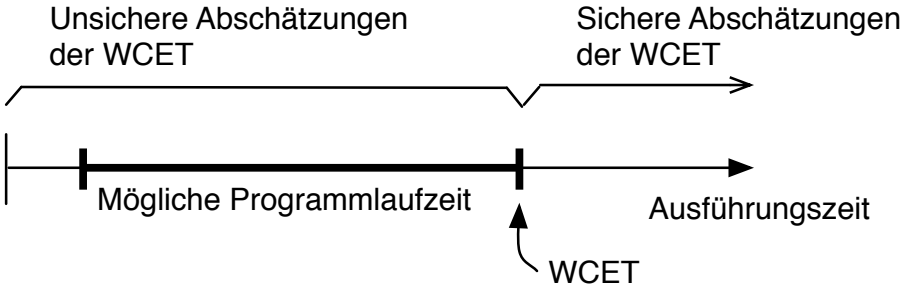

Abbildung 2.1: Abschätzung der WCET

Häufig wird versucht, die WCET durch Messungen in einem Simulator oder auf der tatsächlichen Zielplattform abzuschätzen. Dieses Verfahren wird als dynamische Analyse bezeichnet. Aufgrund der Komplexität moderner Software kann das Verhalten eines Programms aber selten für jede mögliche Eingabe untersucht werden. Da meist nicht bekannt ist, für welche Eingabedaten das ungünstigste Laufzeitverhalten zu erwarten ist, kann so keine sichere Abschätzung der WCET gewährleistet werden. Dies hat zur Folge, dass die eingesetzte Hardware überproportioniert wird, um das Risiko einer falschen Abschätzung zu verringern. Ein solches Vorgehen erhöht natürlich die Kosten, da eine schnellere Hardware in der Regel auch teurer ist. Zudem kann ein korrektes Verhalten des Systems dennoch nicht garantiert werden.

Daher ist insbesondere unter harten Echtzeit-Bedingungen, bei denen die Nichteinhaltung der Zeitschranken zu einer Katastrophe führen kann, eine genauere Kenntnis der WCET unabdingbar. Diese kann durch eine statische Analyse ermittelt werden. Dabei werden keine Messwerte zugrunde gelegt, sondern die Struktur eines Programms analysiert, um so Rückschlüsse auf dessen Verhalten bei beliebigen Eingaben zu erhalten. Auf diese Weise lassen sich verlässliche Werte ermitteln und somit die Sicherheit eines Systems garantieren. Je nach Komplexität der Zielplattform ist die statische Analyse keine leichte Aufgabe: Durch Caches und Befehlspipelines ist schon die Ausführungszeit einer einzelnen Instruktion nicht trivial zu ermitteln und hängt vom Kontext ab, in dem diese Instruktion steht. Ein Cache-Miss beispielsweise kann die Ausführungszeit einer Instruktion um den Faktor 10 oder mehr erhöhen. Damit eine möglichst enge Abschätzung der WCET erreicht wird, darf nicht bei jedem Zugriff von diesem ungünstigen Fall ausgegangen werden. Daraus folgt, dass eine statische Analyse ein differenziertes mathematisches Modell der gesamten Hardware enthalten muss. Zudem wird deutlich, dass die Analyse nur auf dem Maschinencode stattfinden kann, da erst auf dieser Ebene die notwendigen Informationen zur Bestimmung des Hardwareverhaltens vorliegen.

Weiterhin ist die Kenntnis erforderlich, welche Pfade innerhalb eines Programms während der

Ausführung durchlaufen werden können. Durch Schleifen und die bedingte Ausführung einzelner Programmteile ist die Menge der möglichen Pfade für viele Programme enorm groß. Zudem hängt sie von zahlreichen Parametern, beispielsweise dem Wert einer Variablen zu einem bestimmten Zeitpunkt, ab. Aufgrund solcher Abhängigkeiten können nicht alle für eine WCET-Analyse benötigten Kontrollflussinformationen automatisch berechnet werden. Der Programmierer muss diese dann durch entsprechende Annotationen (z.B. der Schleifeniterationsgrenzen) ergänzen. Damit diese Informationen direkt im Programmcode gegeben werden können, müssen sie vom Compiler berücksichtigt, den entsprechenden Maschinenbefehlen zugeordnet, und an die WCET-Analyse weitergereicht werden.

Schließlich ist aus der Menge aller möglichen Kontrollflusspfade derjenige auszuwählen, welcher die höchste Ausführungszeit aufweist. Dieser Pfad wird als Worst-Case Execution Path (WCEP) bezeichnet. Konsequenterweise entspricht die Ausführungszeit dieses Pfades der WCET.

2.2.2 Darstellung des Kontrollflusses von Programmen

Um den Kontrollfluss eines Programms darzustellen, werden häufig Kontrollflussgraphen verwendet. Dies sind gerichtete Graphen, in denen die Knoten ausführbaren Code darstellen und die Kanten den möglichen Programmfluss abbilden. Damit ein solcher Graph nicht unnötig komplex erscheint, kann von von einzelnen Instruktionen abstrahiert und stattdessen Basisblöcke betrachtet werden.

Ein Basisblock ist eine Programmsequenz maximaler Länge ohne weitere Verzweigungen. Das heißt, ein Basisblock kann nur über die erste Instruktion betreten und über die letzte Instruktion verlassen werden. Daher können Basisblöcke ohne Verlust der Genauigkeit als Knoten im Kontrollflussgraphen verwendet werden. Dies macht außerdem die Zuordnung einer maximalen Ausführungszeit zu den Knoten möglich. Wie im vorhergehenden Abschnitt erwähnt, ist die Bestimmung der Ausführungszeit einer einzelnen Instruktion aufgrund von Caches und Pipelines bei komplexeren Architekturen nicht möglich. Dieses Problem betrifft zwar auch die Zusammenhänge zwischen Basisblöcken, da hier jedoch im Allgemeinen mehrere Instruktionen zusammengefasst werden, fällt die Überschätzung der WCET eines Basisblocks deutlich geringer aus. Somit können die WCET-Werte einzelner Kontrollflussknoten durchaus als Grundlage von Optimierungen herangezogen werden.

Abbildung 2.2 zeigt einen simplen Kontrollflussgraph und demonstriert die in dieser Arbeit verwendete Form der Visualisierung. Die Knoten des Graphen stellen Basisblöcke dar, während durch die Kanten der Kontrollfluss zwischen diesen abgebildet wird.

Der Worst-Case Execution Pfad kann in einem solchen Graphen als Folge von Kanten definiert werden, die im ungünstigsten Fall durchlaufen werden. Im Falle von Schleifen kann es passieren, dass Kanten mehrmals durchlaufen werden. Hier ist wie schon erwähnt die Kenntnis der Schleifeniterationsgrenzen nötig. Diese können ebenfalls in den Knoten des Graphen annotiert werden.

2.2.3 Herausforderungen bei der WCET Optimierung

Die meisten Compiler-Optimierungen beschäftigen sich sich mit der Minimierung der durchschnittlichen Programmlaufzeit. Auf diesem Gebiet wurde schon sehr viel Forschung betrieben,

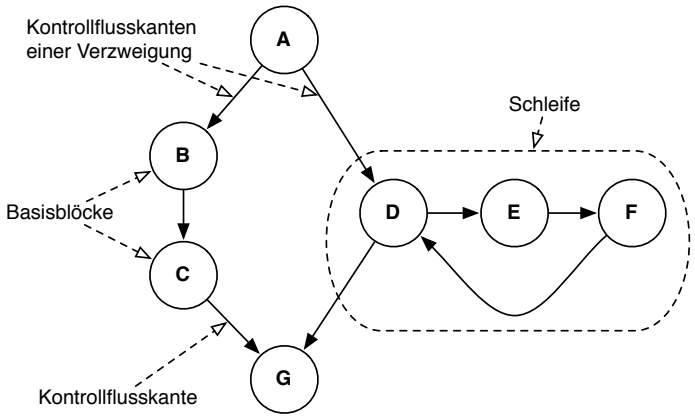

Abbildung 2.2: Beispiel für einen Kontrollflussgraphen

und so sind auch produktiv genutzte aktuelle Compiler wie z.B. der GCC [GNU08b] in der Lage, hochoptimierten Code zu erzeugen [Jon05].

Die Optimierung der Worst-Case Execution Time hingegen ist ein relativ junges Forschungsgebiet, das mit zusätzlichen Herausforderungen aufwartet, die im Folgenden behandelt werden sollen.

Verfügbarkeit von WCET-Informationen

Als Grundlage einer Optimierung werden häufig Informationen über die WCET einzelner Programmabschnitte benötigt. Da Tools zur WCET-Analyse jedoch den kompilierten Code untersuchen, sind diese Informationen zur Übersetzungszeit noch gar nicht verfügbar. Es muss also ein Mechanismus vorhanden sein, um die Ergebnisse der Analyse mit den internen Datenstrukturen der Optimierung bzw. des Compilers zu verknüpfen. Dafür müssen sie gegebenenfalls entsprechend konvertiert werden. Im nächsten Kapitel wird beschrieben, wie dieses Problem in dem für diese Arbeit verwendeten Compiler gelöst wird.

Einfluss lokaler Optimierungen auf die WCET

Der Einfluss von lokalen Optimierungen der WCET kann im Vorfeld nur schwer abgeschätzt werden. Es ist möglich, dass eine Aktion, die lokal eine Verbesserung der Laufzeit bewirkt, sich global negativ auf die WCET auswirkt. Dieser scheinbare Widerspruch ergibt sich ebenfalls aus der Komplexität moderner Hardware. Die Umstellung von Instruktionen innerhalb eines Basisblocks oder das Verschieben von Basisblöcken kann so zum Beispiel zu zusätzlichen Pipeline-Stalls oder Cache Misses führen. Da viele Algorithmen mit mathematischen Modellen arbeiten, in denen solche Effekte nicht berücksichtigt werden können, ist es kaum möglich, solche Anomalien auszuschließen. Bei iterativen Optimierungen müsste nach jedem Schritt eine Neuberechnung der WCET stattfinden. Da eine statische Analyse jedoch für große Programme sehr zeitintensiv ist, ist dieses Vorgehen nicht praktikabel.

Betrachtung des Worst-Case Execution Pfades

Zur Optimierung der durchschnittlichen Laufzeit von Programmen wird oft versucht, die Ausführungszeit besonders häufig ausgeführte Pfade zu verringern. Dies kann unter Umständen negative Einflüsse auf andere, seltener ausgeführte, Pfade haben. Wenn dazu auch der Worst-Case Execution Pfad gehört, was beispielsweise im Fall einer sehr selten ausgeführten, aber kostspieligen bedingten Berechnung nicht unüblich ist, so kann durch eine solche Optimierung die WCET sogar verschlechtert werden. Generell führen Optimierungen außerhalb des Worst-Case Execution Pfades nicht zur Verbesserung der WCET. Daher muss dieser Pfad bekannt sein, wenn es darum geht, beschränkte Ressourcen zuzuteilen (z.B. bei der Registerallokation) oder den Kontrollfluss zu ändern.

Instabilität des Worst-Case Execution Pfades

Das im vorhergehenden Punkt beschriebene Problem wird dadurch verschärft, dass sich der Worst-Case Execution Pfad mit jedem Optimierungsschritt ändern kann. Dies wird in Abbildung 2.3 verdeutlicht. Hier ist ein simpler Kontrollflussgraph dargestellt. In jedem Knoten ist zudem dessen WCET angegeben. Die hervorgehobenen Pfeile kennzeichnen jeweils den WCEP. Die Ausgangssituation ist in (a) illustriert. Kann nun durch eine geeignete Optimierung die WCET des Blocks C auf 15 verringert werden, ändert sich der Worst-Case Execution Pfad wie in (b) dargestellt. Eine weitere Optimierung im Block C wäre nun nicht mehr sinnvoll.

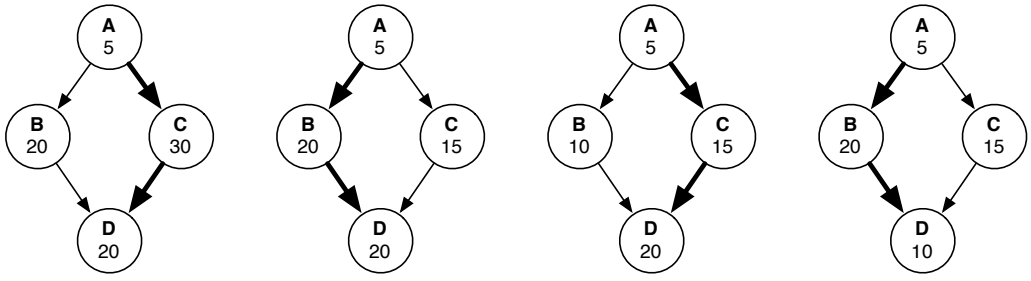

Abbildung 2.3: Instabilität des Worst-Case Execution Pfades

Auch innerhalb des aktuellen Worst-Case Execution Pfades sollte der zu optimierende Block nicht frei gewählt werden, denn jeder Schritt kann Einfluss auf die spätere Optimierung haben. In (c) beispielsweise konnte für den Block B ein Gewinn von 10 Zyklen erreicht werden, wodurch der Worst-Case Execution Pfad wieder wechselt und die WCET auf 40 sinkt. Wird stattdessen, wie in (d) illustriert, der Block D gewählt, kann ebenfalls lokal ein Gewinn von 10 Zyklen erreicht werden. Die WCET des gesamten Programms sinkt durch diese Entscheidung jedoch auf 35.

Es wird deutlich, dass der Instabilität des WCEP in einem Algorithmus zur Minimierung der WCET Beachtung geschenkt werden muss, um optimale Ergebnisse zu erzielen.

2.3 Lineare Optimierungsprobleme

Optimierungsprobleme sind Probleme, die im Allgemeinen viele zulässige Lösungen besitzen. Die Herausforderung dabei besteht darin, dass unter allen möglichen Lösungen diejenige gefunden werden soll, welche die geringsten Kosten (bzw. den höchsten Gewinn) aufweist.

Eine in der Praxis häufig anzutreffende Art von Optimierungsproblemen sind lineare Optimierungsprobleme, auch lineare Programme (kurz LP) genannt. Diese haben eine einfache Struktur: Es soll der Wert einer Zielfunktion minimiert oder maximiert werden, unter Einhaltung von bestimmten Nebenbedingungen, die als lineare Gleichungen gegeben sind. Viele Probleme, insbesondere in der Planung von Produktionsprozessen, werden auf diese Art formuliert und gelöst.

Mathematisch sind lineare Optimierungsprobleme folgendermaßen definiert (aus [MGCK07]): Seien m,n positive ganze Zahlen, $b \in \mathbb{R}^m$ und $c \in \mathbb{R}^n$ Spaltenvektoren mit den zugehörigen Zeilenvektoren b^T und c^T, sowie A eine $m \times n$ Matrix mit Elementen $a_{ij} \in \mathbb{R}$. Eine Instanz eines linearen Optimierungsproblems ist das Problem, einen Vektor $x \in \mathbb{R}^n$ zu finden, der unter allen Vektoren, die die Bedingungen $Ax \geq b$ erfüllen, derjenige ist mit kleinstem (bzw. größtem) Wert $c^T x$. Dabei nennt man die zu minimierende Funktion die *Zielfunktion* des Problems. Die Bedingungen $Ax \geq b$ heißen auch *Restriktionen* oder *Constraints*. Jeder Vektor $x \in \mathbb{R}^n$, der alle Nebenbedingungen erfüllt, heißt *zulässige Lösung*. Die Menge $P = \{x \in \mathbb{R}^d | Ax \leq b\}$ heißt *Zulässigkeitsbereich*.

Kurz lässt sich ein lineares Optimierungsproblem wie folgt schreiben:

$$\begin{aligned} \min\ & c^T x \\ Ax \geq\ & b \\ x \in\ & \mathbb{R}^n \end{aligned}$$

Generell gibt es für den Zulässigkeitsbereich P eines linearen Programms drei verschiedene Möglichkeiten:

1. $P = \emptyset$
   $\Longrightarrow$ Es gibt keine zulässigen Lösungen, somit ist das LP unlösbar.

2. $P \neq \emptyset$, aber das $\inf\{c^T x | x \in P\}$ existiert nicht
   $\Longrightarrow$ Das LP ist lösbar, aber es gibt keine optimale Lösung. Ein solches LP wird als *unbeschränkt* bezeichnet.

3. $P \neq \emptyset$ und das $\min\{c^T x | x \in P\}$ existiert
   $\Longrightarrow$ Das LP ist lösbar, und es gibt eine endliche Lösung x^* mit $c^T x^* = \min\{c^T x | x \in P\}$.

Für lineare Optimierungsprobleme existieren effiziente Algorithmen, so dass auch eine Lösung von Problemen mit mehreren Millionen Variablen möglich ist. Durch die Modellierung eines gegebenen realen Problems als lineares Programm können daher in der Praxis auch sehr komplexe Aufgaben gelöst werden.

Ganzzahlige Lineare Optimierung

Nimmt man die Beschränkung hinzu, dass die Variablen nur ganzzahlige Werte annehmen können, erhöht sich die Komplexität deutlich. Solche Probleme nennt man dann ganzzahlige lineare Programme. Als Abkürzung wird im Folgenden der Begriff *ILP (engl. Integer Linear Program)* verwendet. Wird die Ganzzahligkeit nur für einen Teil der Variablen gefordert, liegt ein gemischt-ganzzahliges lineares Problem vor. Zudem gibt es lineare Programme, deren Variablen nur die Werte 0 und 1 annehmen können. Solche *binären* lineare Programme sind äquivalent zu einem ILP mit den entsprechenden Restriktionen für alle Variablen.

Eine Beschränkung auf ganzzahlige Werte ist in praktischen Problemen sehr gängig: In einem Produktionsprozess können beispielsweise keine halben Autos produziert werden, und Fluggesellschaften können nicht $\frac{5}{3}$ Flugzeuge starten lassen. Auch das bekannte Rucksackproblem lässt sich als ILP formulieren. Hierbei geht es darum, eine Menge von Objekten unter Einhaltung einer Kapazitätsbeschränkung so zu wählen, dass der Gewinn maximiert wird. An diesem Beispiel soll die Formulierung eines Optimierungsproblems als ILP verdeutlicht werden:

Gegeben seien n Objekte mit den Gewichten $g_1, g_2, \ldots, g_n$ und den Nutzwerten $h_1, h_2, \ldots, h_n$. Von diesen Objekten soll eine Teilmenge gewählt werden, so dass der aufsummierte Nutzwert möglichst groß ist, ohne eine Gewichtsbeschränkung G zu überschreiten. Für jedes der Objekte wird eine Variable eingeführt, die entscheidet, ob das entsprechende Objekt gewählt wird. Diese Variablen $x_1, x_2, \ldots, x_n$ können nur die Werte 0 und 1 annehmen, da Objekte nur einmal oder gar nicht mitgenommen werden können. Dies wird durch entsprechende Restriktionen ausgedrückt. Die komplette Formulierung des ILP lautet wie folgt:

$$\max \quad h_1x_1 + h_2x_2 + \cdots + h_nx_n \tag{2.1}$$

$$g_1x_1 + g_2x_2 + \cdots + g_nx_n \leq G \tag{2.2}$$

$$x_i \leq 1 \text{ für } x = 1, 2, \ldots, n \tag{2.3}$$

$$x_i \geq 0 \text{ für } x = 1, 2, \ldots, n \tag{2.4}$$

Die Zielfunktion 2.1 bewirkt die Maximierung des Gesamtgewinns, während Restriktion 2.2 dafür sorgt, dass die Kapazitätsbeschränkung des Rucksacks eingehalten wird. Die übrigen Restriktionen beschränken den Wertebereich aller Variablen auf Werte zwischen 0 und 1. Würde man diese Formulierung als lineares Problem interpretieren, könnten Objekte geteilt und dafür auch der Teilnutzen angerechnet werden. Da es sich jedoch um ein ILP handelt, können die Variablen tatsächlich nur diese beiden Werte annehmen.

Für das Rucksackproblem wird die NP-Vollständigkeit z.B. in [Weg03] bewiesen. Da sich dieses Problem wie gezeigt als ILP formulieren lässt, ist somit auch die ganzzahlige lineare Optimierung NP-schwierig.

Dennoch gibt es Algorithmen, mit denen auch ganzzahlige lineare Programme häufig in vertretbarer Zeit gelöst werden können. Eine Behandlung dieser Algorithmen ist für den Inhalt dieser Arbeit nicht relevant, da es eine Vielzahl an kommerziell und frei verfügbaren Programmen zur Lösung ganzzahliger linearer Optimierungsprobleme gibt. Für weiterführende Informationen sei hier auf [AP01] verwiesen.

Kapitel 3

Framework

Die in dieser Arbeit vorgestellten Optimierungen sind größtenteils architekturunabhängig und damit sehr universell einsetzbar. Die konkrete Implementierung jedoch benötigt viele Eingabewerte, beispielsweise die WCET der einzelnen Basisblöcke und das Datenzugriffsverhalten der Instruktionen. Daher ist es unumgänglich, sich ausführlich mit dem Framework zu beschäftigen, das diese Eingabedaten bereitstellt, und mit dem konkrete Ergebnisse für eine Bewertung der durch den Algorithmus erzielten Auslagerung erlangt werden können. Dieser Abschnitt beschreibt die Zielplattform, das verwendete WCET-Analysetool und natürlich den Compiler, in dem die behandelten Optimierungen implementiert wurden.

3.1 Zielplattform

Die Zielplattform dieser Arbeit ist durch den verwendeten Compiler, den WCET-aware C Compiler (WCC, siehe Abschnitt 3.3) vorgegeben. Der WCC erzeugt Code für den Infineon TC1796. Bei diesem Prozessor handelt es sich um einen 32 Bit RISC-Prozessor der TriCore 1.3 Architektur [Inf02], welcher hauptsächlich für den Einsatz in eingebetteten System konzipiert ist. Der besondere Schwerpunkt liegt hier laut Infineon auf anspruchsvollen Automobil- und Industriesteuerungen.

Der Name TriCore beruht auf dem Ansatz, die Stärke von drei bewährten Architekturen in einem Chip zu vereinen:

- **RISC Mikroprozessor**
  Durch seine *Reduced Instruction Set Computing* Architektur kann der Prozessor mit schnellen Taktraten betrieben werden und stellt eine hohe Anzahl an Registern bereit.
- **DSP**
  Der TriCore bringt eine Reihe von Eigenschaften mit sich, die sonst vor allem bei digitalen Signalprozessoren zu finden sind. Dazu gehören komplexe Adressierungsarten, Multiply-Accumulate Befehle und Schleifen ohne Overhead. Dadurch kann der Prozessor auch rechenintensive Anforderungen sehr performant erfüllen.
- **Mikrocontroller**
  Durch das große Angebot an On-Chip Peripherie (z.B. AD-Wandler, CAN-Interface) kön-

nen viele Aufgaben ohne zusätzliche Hardware erledigt werden. Eine weitere Stärke des TriCore ist die schnelle Behandlung von Kontextwechseln und Interrupts.

Der TriCore implementiert eine Harvard-Architektur mit vier unabhängigen Bussen, je einen für Befehle, Daten und Zugriff auf interne bzw. externe Peripherie. Im Folgenden sollen nun die für diese Arbeit relevanten Eigenschaften des TC1796 beschrieben werden. Für eine ausführliche Beschreibung aller Features sei auf das Datenblatt verwiesen [Inf06].

3.1.1 Befehlssatz und Register

Der TriCore Befehlssatz [Inf08] lässt sich als typischer RISC Befehlssatz mit einigen Erweiterungen, beispielsweise für die DSP-Features und die vereinfachte Unterstützung von Betriebssystemen, klassifizieren. Er besitzt 16 Datenregister, 16 Adressregister, 2 Statusregister und den Programmzähler. Zur Manipulation der Daten- bzw. Adressregister werden unterschiedliche Befehle genutzt. Dies hat die Ursache, dass diese Befehle je auf einer eigenen Pipeline bearbeitet werden (siehe 3.1.2).

Da es sich beim TriCore um eine Load/Store Architektur handelt, operieren die meisten Befehle nur auf Registern. Zusätzlich stehen eine Reihe von Befehlen zum Transport von Daten aus dem und in den Speicher bereit. Diese unterstützen 8 verschiedene Adressierungsarten.

Für viele gängige Befehle stellt die TriCore Architektur 16-Bit Versionen bereit. Dies verringert den Umfang des Codes und hilft somit, Speicher zu sparen.

3.1.2 Pipelines

Beim TriCore handelt es sich um eine superskalare Architektur. Sie besitzt drei Pipelines: Die Integer-Pipeline bearbeitet hauptsächlich arithmetische Instruktionen und bedingte Sprünge, deren Operanden in Datenregistern liegen. Die Load/Store-Pipeline behandelt Speicherzugriffe, Adressarithmetik, unbedingte Sprünge, Funktionsaufrufe und Kontextwechsel. Eine dritte, kleine Pipeline steht zur Ausführung von Zero-Overhead Loops bereit. Bei geschickter Anordnung der Instruktionen können innerhalb eines Zyklus je ein Befehl in die Integer- und in die Load/Store-Pipeline geladen werden. Die Pipelines laufen synchron. Das bedeutet, bei einem Stall werden immer alle Pipelines gleichzeitig gestoppt.

Die beiden Hauptpipelines besitzen je vier Stufen: *Fetch*, *Decode*, *Execute* und *Writeback*. Einzige Ausnahme bildet die Behandlung von Multiply-Accumulate Befehlen. Für diese wird die *Execute*-Stufe der Integer-Pipeline auf zwei Zyklen erweitert. Die Aufgaben der einzelnen Pipelinestufen sind selbsterklärend: In der *Fetch*-Phase wird die nächste auszuführende Instruktion geladen. In der *Decode*-Phase wird sie dekodiert und die benötigten Eingabewerte aus den Registern (oder Zwischenpuffern, die zur Vermeidung von Hazards existieren) geholt. Die *Execute*-Phase führt die Berechnung oder den Speicherzugriff aus, und in der *Writeback*-Phase werden die Ergebnisse in das Zielregister geschrieben.

Ein Überblick über das genaue Verhalten der Pipelines und den Umgang mit Hazards und Stalls findet sich in [Inf04b] und [Inf03].

3.1.3 Speicherarchitektur

Der TC1796 besitzt eine ganze Reihe von On-Chip Speichern, die über verschiedene Busse mit der CPU kommunizieren. Abbildung 3.1 zeigt ein Blockdiagramm des Chips. Die Peripheriebusse inklusive der daran angeschlossenen Einheiten sind aus Gründen der Übersichtlichkeit nicht abgebildet, obwohl auch diese teilweise Zugriff auf die Speicher haben.

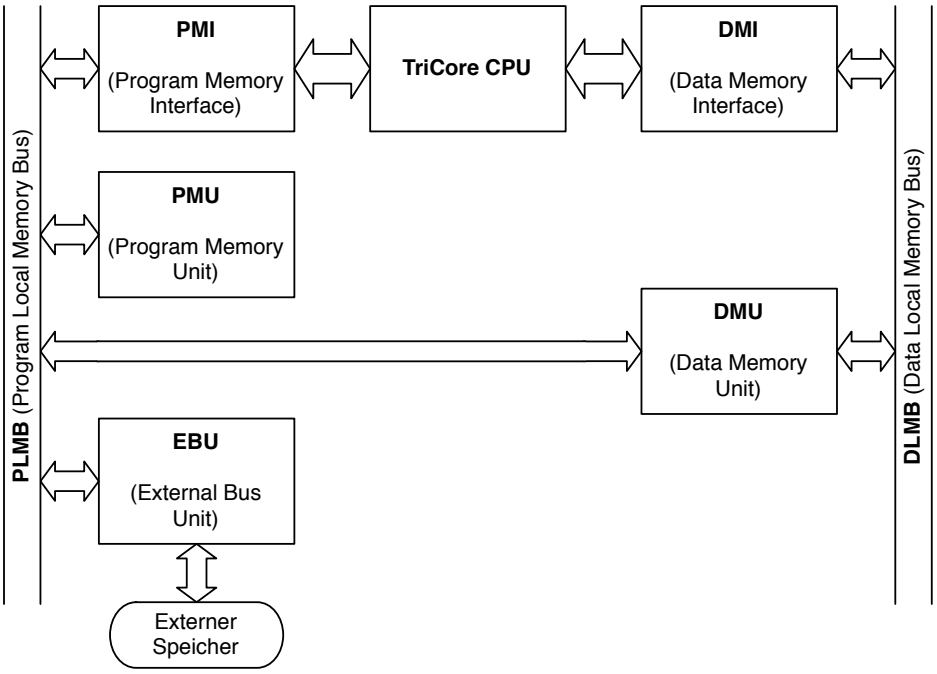

Abbildung 3.1: TC1796 Blockdiagramm

Anhand der Abbildung lässt sich leicht erkennen, dass beim TriCore eine strikte Trennung zwischen Daten und Code herrscht. Dies umfasst unterschiedliche Interfaces, Busse und Speicher. Beide abgebildeten Busse werden mit der Taktrate der CPU betrieben. Sämtliche für diese Arbeit relevanten Speicher auf dem TC1796 befinden sich innerhalb der abgebildeten Einheiten.

- **PMI**
  Das Program Memory Interface ist eng mit der CPU verbunden und enthält neben dem Instruktionscache (16 KByte) auch einen 48 KByte großen Code-Scratchpad. Diese Speicher können von der CPU innerhalb von einem Zyklus angesprochen werden, so dass in der Instruction Fetch Phase der Pipeline in diesen Fällen keine Wartezeit auftritt. Lediglich bei einem Cache-Miss oder dem ungecachten Zugriff auf andere Speicher muss über den Program Local Memory Bus auf diese zugegriffen werden.

- **DMI**
  Das Data Memory Interface ist das Gegenstück zum PMI für Daten. Hier liegt ein 64 KByte großer Scratchpad-Speicher, von dem 8 KByte über einen zweiten Port auch vom externen Peripheriebus (nicht abgebildet) angesprochen werden können. Auch der Zugriff auf diesen Speicher ist für die CPU ohne Wartezeit möglich.

- **PMU**
  Die Program Memory Unit wird vom PMI über den PLMB angesprochen. Sie umfasst neben 16 KByte Boot ROM und 2 MByte Programmflash auch 128 KByte Flash-Speicher für Daten.

- **DMU**
  Auch die Data Memory Unit besitzt einen 64 KByte SRAM. Ein Zugriff von der CPU aus benötigt hier allerdings einige Zyklen. In Kapitel 5.4.2 ist der genaue Ablauf beim Zugriff auf diesen Speicher beschrieben. Zusätzlich enthält die DMU 16 KByte Stand-By RAM, dessen Inhalt auch im Stormsparmodus des Prozessors erhalten bleibt. Ein Zugriff auf weitere Datenspeicher erfolgt über ein Interface zum PLMB, da an diesem sowohl der PMU-interne Datenflash als auch die im nächsten Punkt beschriebene EBU angeschlossen ist.

- **EBU**
  Die External Bus Unit selbst enthält keine Speicher, sondern bietet Zugriff auf externe Speicher. Sie ist an den Program Local Memory Bus angeschlossen und besitzt einen Code Prefetch Buffer. Damit ist die EBU optimiert für den Zugriff auf Programmcode, über das PLMB-Interface der DMU können jedoch auch externe Datenspeicher angesprochen werden.

Die in dieser Arbeit beschriebenen Optimierungen verlagern Daten aus dem SRAM der DMU sowie dem Flash-Speicher der PMU in den schnellen DMI-internen Scratchpad-Speicher, um eine Minimierung der WCET zu erreichen. Da der Scratchpad-Speicher verhältnismäßig groß ist, würde eine funktionierende Optimierung für viele Programme eine Auslagerung aller Datenobjekte veranlassen. Um sinnvolle Ergebnisse zu erzielen, kann daher über die Anpassung einer Konfigurationsdatei des Compilers die Größe des Scratchpads künstlich beschränkt werden.

3.2 aiT

Das Angebot an kommerziell verfügbarer Software zur Analyse der Worst-Case Execution Time ist derzeit noch sehr rar. Ein sehr ausgereiftes Programm ist das von der Firma AbsInt Angewandte Informatik GmbH entwickelte Tool aiT [Abs08a]. Hierbei handelt es sich um ein Tool zur Analyse des Stackverhaltens und der WCET von im Binärcode vorliegenden Programmen. Zur Abschätzung der WCET implementiert aiT ein statisches Analyseverfahren (vgl. Kapitel 2.2.1). Durch die detaillierte Modellierung der Zielarchitekturen inklusive Pipeline- und Cacheverhalten sowie eine sehr differenzierte Betrachtung unterschiedlicher Ausführungskontexte kann aiT die Überschätzung gering halten und sichere Grenzen garantieren.

aiT wurde bisher für sechs unterschiedliche Architekturen angepasst, darunter auch den Infineon TC1796. Dieser Abschnitt vermittelt die nötigen Grundkenntnisse über die Vorgehensweise der WCET-Analyse durch aiT und die Interaktionsmöglichkeiten des Nutzers. Alle Informationen stammen aus dem aiT TriCore Handbuch [Abs08c].

3.2.1 Beschränkungen

Trotz seines großen Funktionsumfangs kann aiT nicht beliebige Programme analysieren. So müssen eine Reihe von Beschränkungen eingehalten werden:

- aiT betrachtet den zu analysierenden Code isoliert. Taskwechsel, Interrupts, die Kommunikation mit Koprozessoren und dergleichen kann daher nicht berücksichtige werden.
- Damit bestimmte Strukturen wie Schleifen erkannt werden, muss der Code in der Programmiersprache C geschrieben und durch einen von aiT unterstützten und in der Konfiguration spezifizierten Compiler übersetzt worden sein.
- Es gibt keine Unterstützung von dynamisch allokiertem Speicher. Dies schließt insbesondere Programme aus, welche die C Funktionen der `malloc`-Familie nutzen.
- Funktionsaufrufe müssen sich an die für die Zielarchitektur spezifizierten Aufrufkonventionen halten. Zudem darf die Rücksprungadresse nicht manipuliert werden.

3.2.2 Control-Flow Representation Language

Während des Analysevorgangs durchläuft aiT eine ganze Reihe unterschiedlicher Tools, die als einzelne Programme vorliegen. Als gemeinsames Ein- und Ausgabeformat verwenden diese Programme die Control-Flow Representation Language (CRL), die mittlerweile in Version 2 vorliegt. Diese umfasst sowohl ein Dateiformat als auch Datenstrukturen zur effizienten Behandlung der Kontrollflussinformationen im Speicher.

Im CRL Format werden Informationen in einer Baumstruktur gespeichert. Einzelne Knoten können Attribute und beliebig viele Unterknoten besitzen. Zusätzlich stellt das Format zahlreiche komplexe Datentypen wie Listen, Schlüssel-Wert Paare und Intervalle zur Verfügung. In dieser Struktur können so zu jedem Element des Kontrollflussgraphen verschiedene Attribute gespeichert werden. Es existiert eine Bibliothek, mit der sich CRL-Dateien importieren, bearbeiten und speichern lassen. Die CRL-Dateien selbst sind simple ACSII-Dateien, die auch mit einem Texteditor verändert werden können. Dabei muss lediglich auf die entsprechende Notation geachtet werden, die in der CRL-Dokumentation beschrieben ist [Abs08b].

In Abbildung 3.2 ist der Aufbau eines CRL-Datensatzes dargestellt. Während die Elemente auf der rechten Seite Einstellungen, Daten und Eigenschaften der Zielarchitektur speichern, wird der tatsächliche Kontrollflussgraph durch die links abgebildeten Elemente gebildet. Ein CFG besteht so aus Routinen (die hauptsächlich zur Darstellung von Funktionen genutzt werden), die wiederum Basisblöcke enthalten. Eine Routine kann in unterschiedlichen Kontexten stehen. Die Attribute aller Unterelemente der Routine können dann für jeden Kontext eigene Werte enthalten. Ein Basisblock speichert neben den Kanten zu seinen Nachfolgern die Instruktionen, welche er enthält. Diese können aus einer oder mehreren Operationen bestehen. Für die TriCore Architektur enthält jede Instruktion nur eine Operation.

Die CRL ist beliebig erweiterbar, was die Integration in eigene Tools ermöglich und durch die in Abschnitt 3.3.3 beschriebene Integration in den WCC ausgenutzt wird. Dennoch ist eine solche Nutzung des Austauschformats eigentlich nicht vorgesehen. Die WCET-Analyse läuft normalerweise komplett durch ein Kontrollprogramm gesteuert ab, das als Eingabe das zu untersuchende

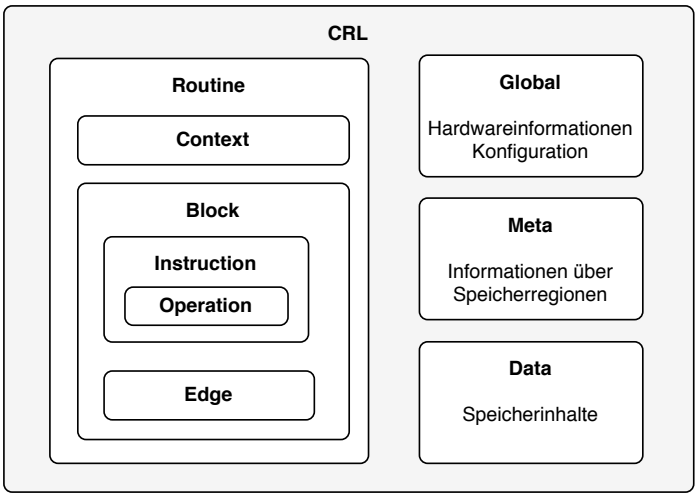

Abbildung 3.2: Struktur der Control-Flow Representation Language

Programm im Binärformat sowie eine Reihe von Konfigurationsdateien erhält. Schließlich ist eine Anzeige des mit WCET-Informationen angereicherten Kontrollflussgraphen mit dem Programm aiSee möglich.

3.2.3 Die Stufen der WCET-Analyse

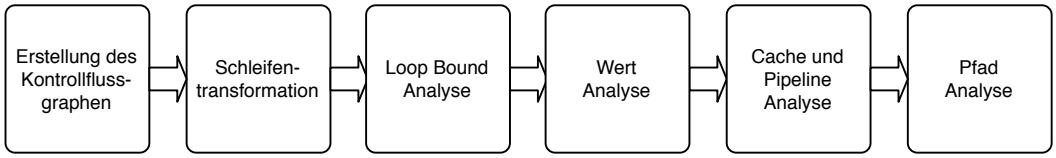

Abbildung 3.3: Die Stufen der aiT WCET-Analyse

Abbildung 3.3 zeigt den Ablauf der statischen WCET-Analyse durch aiT. Jede der abgebildeten Phasen wird durch den Aufruf eines Kommandozeilentools gestartet. Die Eingabe und Ausgabe erfolgt immer als CRL-Datei, die in jedem Schritt transformiert oder mit zusätzlichen Informationen angereichert wird. Die einzelnen Phasen sollen nun kurz beschrieben werden.

Erstellung des Kontrollflussgraphen

Im ersten Schritt wird das zu untersuchende Programm eingelesen und disassembliert. Es wird ein Kontrollflussgraph im CRL-Format erstellt, der den gesamten Programmablauf beschreibt. Dieser umfasst neben den Funktionen (von aiT Routinen genannt), Basisblöcken, Instuktionen und Operationen auch die globalen Daten, also den Speicherinhalt zum Start des Programms. Auch die in Konfigurationsdateien spezifizierten Informationen werden in der CRL Datei gespeichert. Dazu gehören die Benutzerannotationen (siehe Abschnitt 3.2.4) und Angaben über die Hardware, z.B. die Taktrate oder das Zugriffsverhalten externer Speicher.

Schleifentransformation

Um genauere WCET-Abschätzungen für eine Funktion zu berechnen, kann aiT den Kontext unterscheiden, in dem eine Funktion aufgerufen wurde. Dies ist ein sehr sinnvoller Mechanismus, denn eine Funktion kann, wenn sie von unterschiedlichen Stellen betreten wird, ein sehr unterschiedliches Verhalten aufweisen. Dies gilt eingeschränkt auch für Schleifen. Die erste Iteration einer Schleife beispielsweise kann eine sehr viel höhere Laufzeit als die darauf folgenden haben, da zunächst der Schleifencode in den Cache geladen wird. Um das Konzept der Kontexte auch für Schleifen nutzen zu können, konvertiert aiT daher jede Schleife in einen rekursiven Funktionsaufruf.

Loop Bound Analyse

In der dritten Phase versucht aiT, Schleifeniterationsgrenzen (Loop Bounds) automatisch zu ermitteln. Dies funktioniert jedoch nur für sehr simple Schleifen, so dass meist eine manuelle Annotation durch den Benutzer erforderlich ist.

Wertanalyse

Die Wertanalyse berechnet mögliche Registerinhalte für jede Programmstelle. Mit Hilfe dieser Werte können unmögliche Pfade identifiziert und die Ziele von Speicherzugriffen berechnet werden. Häufig können hier keine exakten Werte berechnet werden, so dass aiT mit Intervallen von möglichen Werten rechnet. Damit von dem bereits erwähnten Konzept der Kontexte profitiert werden kann, muss die Wertanalyse jeden Kontext gesondert betrachten. Dadurch ergibt sich besonders in dieser Phase ein sehr hoher Rechenaufwand, der sich in der benötigten Zeit der WCET-Berechnung bemerkbar macht.

Cache und Pipeline Analyse

In dieser Phase wird unter Berücksichtigung der möglichen Cache- und Pipelinezustände für jede Kante des Kontrollflussgraphen die WCET bestimmt. Die für eine Kante $A \rightarrow B$ berechnete WCET beschreibt eine obere Grenze der Anzahl an Zyklen, die eine Ausführung des Blockes A benötigt, wenn der Kontrollfluss danach am Block B fortgesetzt wird. Dies umfasst lediglich die in A selbst enthaltenen Instruktionen, nicht die Laufzeit von eventuell in A aufgerufenen Funktionen.

In dieser Phase werden die in der vorherigen Stufe berechneten Registerinhalte genutzt, um für Load/Store Instruktionen die unterschiedlichen Zugriffszeiten der Speicher zu berücksichtigen.

Pfadanalyse

Der Pfadanalyse stehen nun alle Informationen über die WCET an den Kanten des Kontrollflussgraphen und unmögliche Pfade zur Verfügung. Daraus wird ein ILP-Modell erstellt, das durch einen externen LP-Solver gelöst wird. Aus der Lösung des ILP kann aiT nun die WCET des gesamten Programms und den Worst-Case Execution Pfad ablesen.

Zusätzlich zur graphischen Anzeige des annotierten Kontrollflussgraphen können auf Wunsch auch alle für den Benutzer relevanten Informationen in einer Textdatei abgelegt werden.

3.2.4 Möglichkeiten der Einflussnahme durch den Nutzer

Der Benutzer hat die Möglichkeit, aiT bei bestimmten Berechnungen durch Annotationen zu unterstützen. Diese werden in einer Konfigurationsdatei mit der Endung `.ais` notiert und an aiT Übergeben. Der `exec2crl`-Konverter baut dann diese Informationen in die CRL-Struktur ein. Da jeder Schritt der Analyse auch einzeln ausgeführt werden kann, können durch Editieren der CRL-Dateien auch zwischen den Schritten Informationen geändert oder hinzugefügt werden.

Einige Annotationen wie die Angabe von nicht automatisch erkannten Loop Bounds sind nötig, damit aiT korrekt funktioniert, andere können die WCET-Abschätzung verbessern oder die Laufzeit der Analyse verringern. Zu letzterem Zweck können beispielsweise global oder pro Routine die Speicherbereiche definiert werden, auf welche ein lesender bzw. schreibender Zugriff erfolgt. Damit lässt sich die Anzahl an möglichen Werten in der Wertanalyse verringern. Zudem kann so vermieden werden, dass aiT für die entsprechenden Load/Store Befehle vom ungünstigsten Fall, also dem Zugriff auf den langsamsten Speicher, ausgeht.

Insbesondere lassen sich auch pro Instruktion die möglichen Ziele eines Speicherzugriffs angeben. Diese Methode wird in dieser Arbeit genutzt, um die teilweise sehr ungenaue Wertanalyse von aiT für im Compiler bekannte Zugriffsziele zu unterstützen.

In einer zweiten Konfigurationsdatei können aiT genauere Informationen über die verwendete Hardware zur Verfügung gestellt werden. So lassen sich beispielsweise die Eigenschaften von externen Speichern, die Startadresse des Stack Pointers oder die Taktrate des Systems angeben.

3.3 WCC

Der WCET-aware C Compiler (WCC) ist ein am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelter Compiler, der Code für die Infineon TriCore Architektur erzeugt [FLT06]. Der Schwerpunkt liegt, wie der Name andeutet, in der Analyse und Optimierung der Worst-Case Execution Time der zu übersetzenden Programme. Daher integriert der Compiler nicht nur zahlreiche WCET-zentrierte Optimierungen, sondern auch das statische WCET-Analyse Tool aiT, das im Abschnitt 3.2 beschrieben wurde.

Der WCC unterstützt die direkte Erzeugung von Code aus Quelltextdateien im C99 Standard [ISO07]. Auf Wunsch können detaillierte Informationen über die WCET des Programms und der einzelnen Funktionen erzeugt werden. Durch die enge Integration von aiT können auch im Compiler implementierte Optimierungen diese Daten nutzen und so gezielt entlang des Worst-Case Execution Pfades optimieren.

Im WCC werden zahlreiche am Lehrstuhl und am Informatik Centrum Dortmund (ICD) [ICD08] entwickelte Bibliotheken und Optimierungen unter einem homogenen Interface vereint. Die für diese Arbeit relevanten Komponenten des WCC werden in diesem Unterkapitel beschrieben.

3.3.1 Aufbau des Compilers

Der Aufbau des WCC (siehe Abbildung 3.4) entspricht dem typischen Design eines Compilers [GE99]. Der Übersetzungsvorgang beginnt im so genannten Compiler Frontend. Hier geschehen alle architekturunabhängigen Transformationen und Optimierungen auf dem zu übersetzenden Programm.

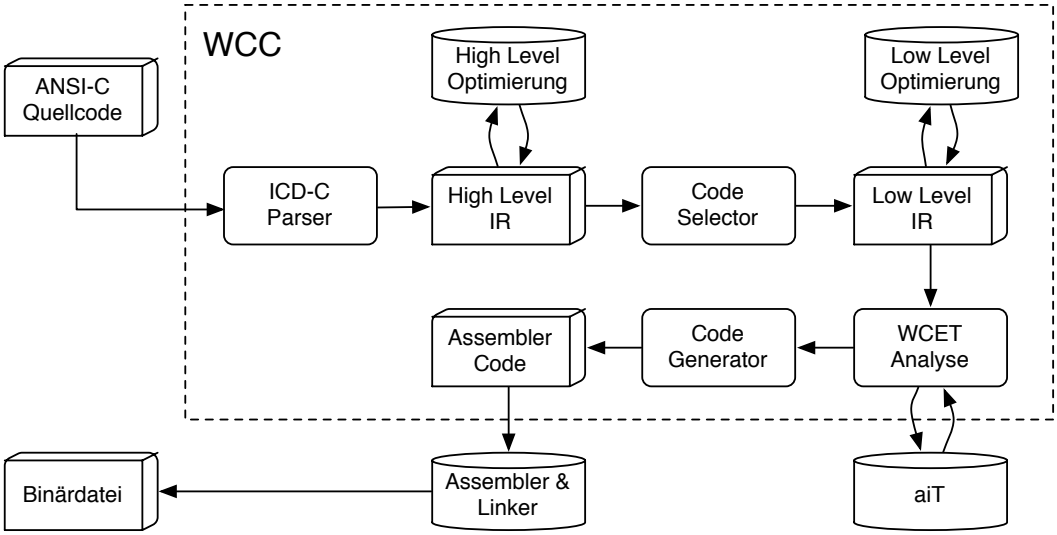

Abbildung 3.4: Aufbau des WCC

Zunächst müssen die ASCII-Symbole der Quelldatei als Symbole der Zielsprache erkannt (lexikalische Analyse) und unter Berücksichtigung der Grammatik der Sprache ein Syntaxbaum des Programms erstellt (Syntaxanalyse) werden. Ist diese Konvertierung nicht erfolgreich, liegt ein Syntaxfehler im Code vor und die Kompilierung wird mit einer entsprechenden Fehlermeldung abgebrochen. Im nächsten Schritt findet die semantische Analyse statt, in der dem Syntaxbaum semantische Informationen hinzugefügt werden. In dieser Phase können (semantische) Fehler oder Unklarheiten im Code festgestellt werden, die mit einem Abbruch oder der Ausgabe einer Warnung behandelt werden. Schließlich kann daraus eine Zwischenrepräsentation (engl. *Intermediate Representation, IR*) erstellt werden, auf der die folgenden Komponenten des Compilers arbeiten.

Alle diese Aufgaben übernimmt im WCC der ICD-C Parser, die entsprechende Zwischendarstellung heißt ICD-C [ICD05]. ICD-C bietet über ein objektorientiertes Interface Zugriff auf die Elemente des Programms. So können auf einfache Weise Optimierungen auf hoher Ebene durchgeführt werden. ICD-C bewegt sich nah am C-Standard, es existierten beispielsweise Klassen zur Beschreibung von Funktionen, Statements, Operationen oder Datentypen.

Die optimierte IR aus dem Frontend wird nun im Compiler Backend weiterbehandelt, in dem die architekturabhängigen Optimierungen und schließlich die Synthese des Zielcodes stattfinden. Dazu muss die High-Level IR zunächst in eine hardwarenahe Low-Level Repräsentation übersetzt werden. Dies geschieht durch den Code Selector, der zu dem im Syntaxbaum definierten Programm die passenden Maschinenbefehle wählt. Die im WCC verwendete Low-Level Repräsentation heißt ICD-LLIR und wird im nächsten Abschnitt ausführlicher behandelt.

Die LLIR nutzt zunächst so genannte virtuelle Register. Dies ermöglicht eine erste Phase an Optimierungen ohne Rücksichtnahme auf die Beschränkung der Registeranzahl der Zielarchitektur. Daraufhin müssen die virtuellen Register auf den realen Registersatz übertragen werden. In diesem *Registerallokation* genannten Schritt kann es vorkommen, dass die Anzahl an Registern nicht ausreicht und so Daten zeitweise auf dem Stack zwischengespeichert werden müssen (*spilling*). Schließlich kann eine zweite Phase an Optimierungen auf der LLIR ausgeführt werden.

Damit ist die Struktur des finales Programms festgelegt. An dieser Stelle kann also die WCET bereits bestimmt werden. Dazu muss der Code jedoch im Eingabeformat von aiT vorliegen und dessen Ausgabe wieder vom Compiler interpretiert werden. Diese Schritte werden in Abschnitt 3.3.3 beschrieben. Nach der WCET-Analyse können natürlich weitere Optimierungen vorgenommen werden, welche die entsprechenden Daten nutzen.

Schließlich wird aus der Low-Level Repräsentation Assemblercode erzeugt, welcher mit Hilfe des GNU Assemblers und Linkers für die Tricore-Architektur [GNU08b, GNU08a] zu einer ausführbaren Datei übersetzt wird.

3.3.2 Low-Level Intermediate Representation

Die *Low-Level Intermediate Representation (LLIR)* ist die im WCC verwendete Zwischendarstellung zur Speicherung und Manipulation der Programmobjekte nahe der Maschinenebene. Die LLIR wurde am Informatik Centrum Dortmund (ICD) [ICD08] entwickelt und besteht aus einer plattformunabhängigen Klassenbibliothek, die an beliebige Architekturen angepasst werden kann.

Jede LLIR-Instanz repräsentiert eine Kompilationseinheit des Eingabecodes. Diese besteht aus Funktionen, welche wiederum eine Sequenz von Basisblöcken enthalten. Jeder dieser Basisblöcke speichert seine Vorgänger und Nachfolger und die in ihm enthaltenen Instruktionen. Zur Unterstützung von VLIW-Architekturen kann eine Instruktion aus mehreren Operationen bestehen. Für den TC1796 ist dies nicht möglich, daher besteht dort zwischen diesen Elementen eine $1:1$-Beziehung.

Abbildung 3.5 zeigt ein Klassendiagramm mit den wichtigsten Komponenten der LLIR. Der besseren Lesbarkeit und Wiedererkennung halber wurden die originalen Klassennamen leicht geändert. Eine ausführliche Dokumentation der Bibiliothek findet sich in [ICD07].

Instruktionen und Operationen

Die LLIR speichert Instruktionen innerhalb eines Basisblocks in einer Liste. Die Reihenfolge in dieser Liste entspricht dabei dem späteren Programmablauf. Jede Instruktion besteht aus mindestens einer Operation, die jeweils das direkte Äquivalent zu einem Assemblerbefehl ist. Daher werden dort neben der eindeutigen Bezeichnung des Maschinenbefehls auch die Parameter gespeichert. Es gibt unterschiedliche Typen von Parametern, die je nach Operation genutzt werden können:

- **Konstanten**
  Mit Konstanten lassen sich unveränderliche numerische Werte übergeben, beispielsweise beim Addieren eines festen Wertes zu einem Register.

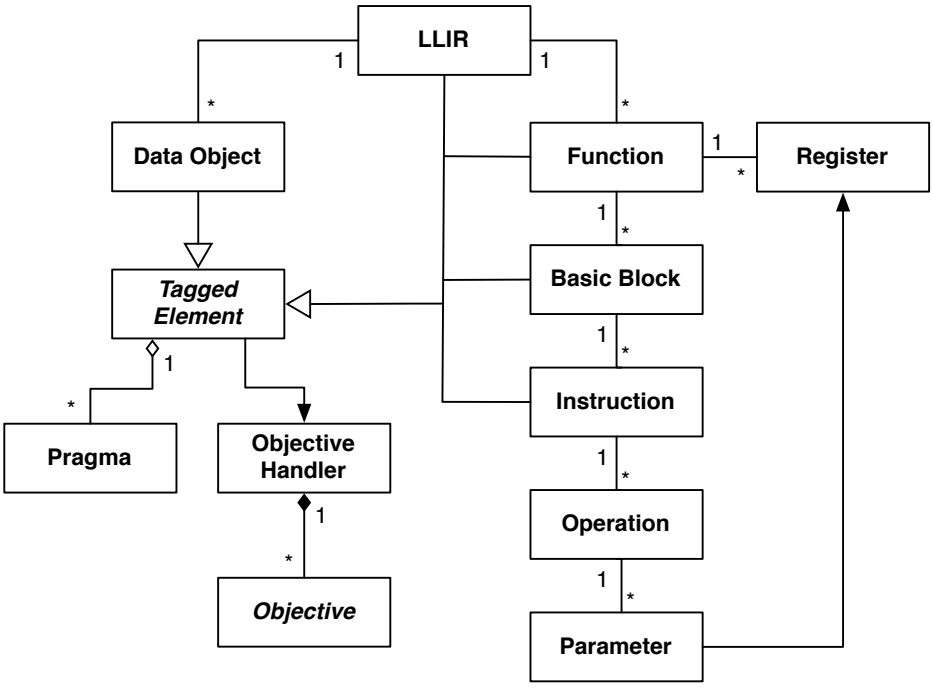

Abbildung 3.5: Klassendiagramm der LLIR

- **Labels**
  Labels referenzieren eine Adresse im Assemblercode unter einem Namen. Dies wird beispielsweise für Sprungziele oder Adressen von Daten genutzt. Aus Sicht der LLIR sind Labels lediglich Zeichenketten, es existiert keine direkte Verbindung zu dem von einem Parameter referenzierten Element.

- **Operatoren**
  Operatoren sind keine tatsächlichen Parameter, sondern sie beziehen sich auf einen anderen Parameter und ergänzen diesen um zusätzliche Informationen für die Operation. Dadurch können beispielsweise Adressierungsarten eines Load/Store-Befehls angegeben werden oder die automatische Inkrementierung eines Parameters nach der Operation angeordnet werden, falls die Architektur dies unterstützt.

- **Register**
  Register sind der einzige Typ von Parametern, der durch eine eigene Klasse beschrieben wird. Dies ermöglicht die Verwaltung von Registern auf Funktionsebene: Jede Funktion speichert alle in ihr genutzten Register, so dass gegenseitige Abhängigkeiten erkannt und freie Register identifiziert werden können. Es wird zwischen virtuellen und physikalischen Registern unterschieden, wobei nur letztere ein Gegenstück auf Maschinenseite haben. Solange virtuelle Register in einer LLIR genutzt werden, kann daraus kein Assemblercode generiert werden.

Manipulation des Kontrollflusses

Der Kontrollfluss eines Programms kann in der LLIR anhand der Basisblöcke verfolgt werden. Jeder Basisblock speichert die Liste seiner Vorgänger und Nachfolger innerhalb der Funktion. Hierbei ist bei Transformationen darauf zu achten, dass diese Informationen mit dem tatsächlichen Verhalten der im Basisblock gespeicherten Instruktionen übereinstimmt. Insbesondere dürfen Kontrollflussverzweigungen nach Definition nur am Ende eines Blocks erfolgen. Sprungbefehle können sich daher immer nur innerhalb der letzten Instruktion befinden.

Jeder Basisblock besitzt ein Label, dessen Eindeutigkeit von der LLIR sichergestellt wird. Über dieses kann er von Operationen aus referenziert und angesprungen werden.

Speicherung von Zusatzinformationen

Wie anhand des Klassendiagramms erkennbar ist, erben eine ganze Reihe von Klassen der LLIR-Bibliothek von *Tagged Element*. Diese abstrakte Klasse bietet die Möglichkeit, zusätzliche Informationen über das Assemblerprogramm zu speichern. Dazu gehören beispielsweise die Namen der Quelldateien oder Zeilennummern, die beim Debuggen hilfreich sein können. In einer Liste von Pragmas können zudem beliebige weitere Daten als String gespeichert werden.

Für aufwändige Optimierungen kann es oft nötig sein, sehr viele und differenzierte Daten zu einem LLIR-Element zu speichern. Für diese Fälle ist eine Speicherung in Strings nicht sinnvoll. Daher ist es möglich, eigene Datenstrukturen an jedes *Tagged Element* zu hängen, ohne die LLIR direkt erweitern zu müssen. Zu diesem Zweck existiert der *Objective Handler*, der den Zugriff auf diese Daten verwaltet. Hier können beliebige von *Objective* erbende Klassen registriert und später abgefragt werden. So ist eine Speicherung von Zusatzinformationen zu LLIR-Elementen möglich, ohne die Bibliothek selbst für jeden möglichen Einsatzzweck anzupassen.

Darstellung von Datenobjekten

Globale und statische Variablen eines Programms sind nicht an eine bestimmte Funktion gebunden. Daher müssen diese auch innerhalb der LLIR über das oberste Element der Klassenhierarchie erreichbar sein. Vor dem Beginn dieser Arbeit existierten in der LLIR keine Datenstrukturen zur Speicherung dieser Daten. Dennoch mussten im WCC die entsprechenden Assemblerdirektiven erzeugt werden. Als Ausweg wurden diese im Codeselektor als Pragmas an die LLIR gehängt und später in die Zieldatei geschrieben.

Es wurde schnell deutlich, dass die Implementierung einer Scratchpad-Allokation ein differenzierteres Modell für Daten benötigen würde, das neben dem Label auch Größe, Inhalt und technische Informationen wie beispielsweise die Ausrichtung an Speichergrenzen speichern kann. Daher wurde die LLIR um das Konzept von Datenobjekten erweitert. Die Implementierung dieser Erweiterung ist jedoch abseits des Inhalts dieser Arbeit und soll daher nicht ausführlicher behandelt werden.

3.3.3 Integration der WCET-Analyse

Besonderes Merkmal des WCC ist die enge Integration des statischen WCET-Analyse Tools aiT, das im Abschnitt 3.2 bereits beschrieben wurde. Dieses ist eigentlich für die Analyse von ausführbaren Dateien der Zielarchitektur vorgesehen, die durch den Compiler erst im letzten Schritt erzeugt werden. Eine weitere Nutzung der erzeugten WCET-Daten wäre so nicht möglich. Es wurde daher ein Mechanismus geschaffen, um diese Informationen direkt auf LLIR-Ebene verfügbar zu machen. Dazu mussten mehrere Schritte vollzogen werden, die Lokuciejewski in seiner Diplomarbeit detailliert beschreibt [Lok05]. Der Ablauf der WCET-Analyse im WCC ist in Abbildung 3.6 dargestellt.

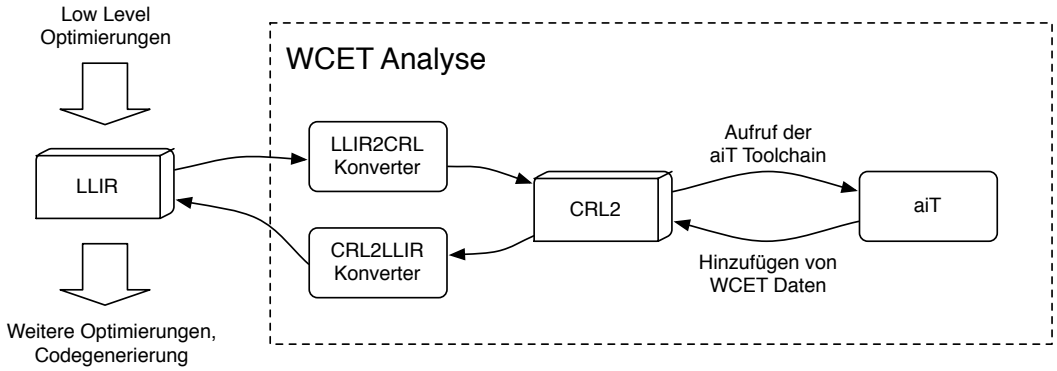

Abbildung 3.6: Integration der WCET-Analyse in den WCC

Zunächst muss das zu analysierende Programm als eine oder mehrere LLIR-Instanzen ohne virtuelle Register vorliegen. Erst in dieser Form stehen der hardwarenahen WCET-Analyse alle Daten zur Verfügung, die benötigt werden, um zyklengenaue Berechnungen durchzuführen. Dazu muss das Programm jedoch zunächst im Eingabeformat von aiT, der Control-Flow Representation Language, vorliegen. Im Aufbau der LLIR und der CRL gibt es sehr viele Ähnlichkeiten, so dass eine Umwandlung überhaupt erst möglich ist. Nach dem Aufruf der aiT-Toolchain müssen dann die erzeugten WCET-Informationen zurück in die LLIR übertragen werden, wodurch sie von späteren Phasen das Compilers genutzt werden können. Selbstverständlich können auch mehrere Durchläufe der WCET-Analyse pro Kompilierungsvorgang durchgeführt werden, was auch iterative Optimierungen und Zwischenmessungen möglich macht.

Für die Überführung der beiden Low-Level Darstellungen ineinander musste einiger Aufwand betrieben werden. Die folgende Beschreibung der Konverter soll daher nur einen Überblick geben und erhebt keinen Anspruch auf Vollständigkeit.

Konvertierung der LLIR in das CRL-Format

Normalerweise erstellt aiT die während der statischen Analyse bearbeitete CRL selbst aus einer Binärdatei. Innerhalb des WCC wird diese Struktur direkt aus der LLIR erzeugt. Dies geschieht durch einen Konverter, der neben der Übersetzung des Kontrollflussgraphen auch die sonst in einer Konfigurationsdatei spezifizierten Informationen in das CRL-Format überträgt. Das Resultat ist eine zu aiT kompatible CRL-Datei, in der durch Markierung der Basisblöcke jeder CRL-Block seinem Äquivalent in der LLIR zugeordnet werden kann.

Die beiden Darstellungen sind jedoch nicht komplett äquivalent. So stehen im CRL-Format insbesondere die Adressen von Daten und Instruktionen fest, während diese in der LLIR noch gänzlich unbekannt sind. Zudem wird eine LLIR-Operation durch einen Assemblerbefehl und eine Liste von Operanden dargestellt, während auf CRL-Ebene Maschinenbefehle eindeutig identifiziert werden. Es können jedoch für eine LLIR-Operation abhängig von der Parameterliste unterschiedliche Maschinenbefehle benötigt werden, so dass der Konverter an dieser Stelle eine entsprechende Auswahl vornehmen muss.

Schließlich können die im Abschnitt 3.2.3 beschriebenen Schritte der Analyse nacheinander ausgeführt werden.

Anreichern der LLIR mit WCET-Informationen

Nachdem aiT die WCET-Analyse beendet hat, stehen die entsprechenden Informationen in der CRL bereit. Diese müssen nun den zugehörigen LLIR-Elementen zugeordnet werden. Diese Aufgabe übernimmt ein zweiter Konverter. Dank der im ersten Schritt vorgenommenen eindeutigen Kennzeichnung von Basisblöcken ist eine Identifizierung von zusammengehörigen Blöcken leicht möglich.

Die Speicherung aller von aiT berechneten Informationen geschieht durch die Nutzung des *Objective Handler*. In der hier registrierten Klasse *LLIR_WCET_OBJ* werden nach der Konvertierung für jeden Basisblock die von aiT ermittelten Daten gespeichert. aiT speichert jedoch WCET-Informationen auch an Kanten zwischen den Blöcken. Diese sind in der LLIR nur implizit durch die Speicherung von Vorgänger und Nachfolger pro Block verfügbar und können somit nicht selbst mit zusätzlichen Daten versehen werden. Daher wird im *LLIR_WCET_OBJ* auch eine Liste von Kanten verwaltet, für die jeweils die WCET gespeichert wird. An dieser Stelle werden auch Informationen über unterschiedliche Kontexte gespeichert, da die LLIR hierfür kein vergleichbares Konzept bietet.

Neben der WCET stehen nach der Konvertierung auch Informationen über die Unausführbarkeit oder Ausführungshäufigkeiten der Blöcke und Kanten bereit. All diese Daten können dann durch folgende Komponenten des Compilers genutzt werden.

Kontrollflussinformationen

Die beschriebene enge Integration der WCET-Analyse in den Compiler birgt neben der Verfügbarkeit von WCET-Daten im Compiler auch den Vorteil, dem Nutzer den manuellen Aufruf von aiT zu ersparen. In der beschriebenen Form hat dies jedoch noch einen Schönheitsfehler: Die Angabe von Kontrollflussinformationen wie z.B. Schleifeniterationsgrenzen müsste so immer noch auf Basis der Low-Level Repräsentation stattfinden. Diese sollte im Normalfall jedoch nicht von Interesse für den Nutzer sein. Zudem kann durch Optimierungen wie Schleifentransformationen die Struktur der LLIR deutlich vom ursprünglichen Eingabecode abweichen. Die Annotation auf dieser Ebene würde also für den Nutzer ein aufwändiges und fehleranfälliges Verfahren darstellen.

Um dies zu verhindern, wurde eine Möglichkeit geschaffen, Kontrollflussinformationen direkt im Quellcode anzugeben [Sch07]. Durch die Integration des Konzepts von *Flow Facts* in den WCC können solche Informationen auf Hochsprachenebene angegeben werden. Es ist dann die

Aufgabe des Compilers, diese bei allen Kontrollflusstransformationen zu berücksichtigen, so dass während der WCET-Analyse alle CRL-Elemente korrekt annotiert werden können.

Der WCC bietet so die optimale Umgebung für die Implementierung und Evaluierung von WCET-zentrierten Optimierungen.

Kapitel 4

ILP-Formulierung der statischen Scratchpad-Allokation

Dieses Kapitel beschreibt den Algorithmus zur statischen Auslagerung von globalen Daten in den Scratchpad Speicher. Der Algorithmus ist unabhängig von der Zielarchitektur. Alle implementierungsspezifischen Details werden erst im nächsten Kapitel behandelt.

Zunächst soll die Problemstellung erläutert werden. Daraufhin wird auf die Grundlagen des Algorithmus eingegangen und nacheinander die Behandlung verschiedener Kontrollflussstrukturen erläutert. Schließlich werden die Einschränkungen beschrieben, die sich aus der ILP-Formulierung ergeben.

4.1 Problemstellung

Zunächst soll kurz die genaue Problemstellung erläutert werden. Gegeben ist der Kontrollflussgraph des zu optimierenden Programms. Es wird vorausgesetzt, dass bekannt ist, welche Instruktionen auf welche Daten zugreifen, und die Basisblöcke besitzen Informationen über ihre maximale Ausführungszeit. Zudem ist natürlich die Größe der einzelnen Datenobjekte und des Scratchpad-Speichers bekannt. Die Größe des Hauptspeichers wird als ausreichend angenommen, um alle Daten des Programms zu fassen. Zusätzlich ist die Kenntnis der Geschwindigkeiten der Speicher notwendig. Genauer muss die Anzahl an Zyklen bekannt sein, die ein einzelner Zugriff auf den Scratchpad-Speicher bzw. den SRAM benötigt.

Zu bestimmen ist nun die Menge derjenigen Datenobjekte, die statt im Hauptspeicher im Scratchpad-Speicher gelagert werden, um eine minimale Worst-Case Execution Time des gesamten Programms zu erreichen. Diese Menge wird statisch bestimmt, das heißt, der Scratchpad-Inhalt ändert sich während des Programmverlaufs nicht.

In dieser Arbeit werden ausschließlich globale Daten betrachtet. Das sind solche, deren Variablen im Quellcode global oder statisch deklariert sind. Die Anwendbarkeit des Algorithmus ist allerdings nicht auf diese Typen von Datenobjekten beschränkt. Für den Algorithmus spielt es keine Rolle, ob eine Variable lokal an eine Funktion gebunden oder global zugreifbar ist, solange die Größe und die Zugriffszeit bekannt sind (für rekursive Funktionen bestehen hier weitere Einschränkungen, die aber nicht näher erläutert werden sollen). Die Unterschiede ergeben sich

erst in der Umsetzung der Auslagerung, da lokale Variablen typischerweise auf dem Stack gespeichert werden, globale Daten jedoch während der Laufzeit des Programms eine feste Adresse im Speicher besitzen.

Zu beachten ist bei der Optimierung insbesondere die Instabilität des Worst-Case Execution Pfades: Wird ein Objekt ausgelagert, weil auf diese Weise für einen Basisblock der größte Gewinn erzielt werden kann, so hat dies Einfluss auf alle anderen Stellen, an denen auf dieses Datenobjekt zugegriffen wird. Zudem nimmt es natürlich Platz im Scratchpad ein, so dass eventuell andere Objekte dann nicht mehr hinein passen. Bei einem Greedy-Vorgehen würde nur jeweils entlang des Worst-Case Execution Pfades optimiert. Dies könnte dazu führen, dass zu Beginn Objekte dem Scratchpad-Speicher zugewiesen werden, die in den folgenden Schritten den Platz für eventuell günstigere Auslagerungen belegen. Eine Optimierung muss alle Auslagerungen global und gleichzeitig betrachten, um nicht solche Effekte aufzuweisen. Daher wurde für die Implementierung ein ILP-basiertes Verfahren gewählt, wodurch alle Nebeneffekte implizit mit betrachtet werden.

Das in diesem Kapitel vorgestellte Verfahren basiert auf einer von Suhendra et al. in [SMRC05] behandelten Methode, wurde jedoch in wesentlichen Punkten erweitert, um eine praktische Anwendung für beliebige Programme zu ermöglichen. Insbesondere eine Unterstützung von Funktionsaufrufen wird in der o.g. Arbeit nicht behandelt.

4.2 Funktionsweise des Algorithmus

Der Algorithmus formuliert das Problem der Scratchpad-Allokation als ganzzahliges lineares Optimierungsproblem (siehe Kapitel 2.3), so dass die WCET des Programms unter Beachtung einiger Nebenbedingungen minimiert wird. Dafür ist es nötig, eine Reihe von Variablen und Konstanten zu definieren. Um eine einfache Unterscheidbarkeit zu gewährleisten, werden Variablen im Folgenden durch einen Großbuchstaben bezeichnet, die Namen von Konstanten beginnen mit einem kleinen Buchstaben.

Die Menge der globalen Datenobjekte eines Programms sei $data\_objects$. Für jedes Datenobjekt $d \in data\_objects$ soll durch das ILP ein Speicherort bestimmt werden. Dafür werden die binären Entscheidungsvariablen S_d mit folgender Bedeutung eingeführt:

$$S_d = \begin{cases} 1 & \text{falls } d \text{ im Scratchpad liegt} \\ 0 & \text{sonst} \end{cases} \tag{4.1}$$

Dazu muss durch Nebenbedingungen sichergestellt werden, dass die Variablen S_d tatsächlich nur die Werte 1 und 0 annehmen können.

$$S_d \geq 0 \qquad S_d \leq 1 \tag{4.2}$$

Durch Einführung der Konstanten $scratchpad\_size$ für die Größe des Scratchpad-Speichers und $size_d$ für die Größe der einzelnen Datenobjekte kann dann die Kapazitätsbeschränkung als Constraint formuliert werden:

$$\sum_{d \in data\_objects} S_d * size_d \leq scratchpad\_size \tag{4.3}$$

4.3 Modellierung von Basisblöcken und Datenzugriffen

Die Worst-Case Execution Time eines Programms lässt sich aus den Ausführungszeiten der Basisblöcke auf dem ungünstigsten Pfad berechnen. Daher muss für jeden Basisblock eine obere Schranke für dessen Ausführungszeit bekannt sein. Diese stellt eine Eingabe des Algorithmus und somit eine Konstante in der Formulierung als lineares Optimierungsproblem dar. Da die Ausführungszeit eines Blocks aber von dessen Datenzugriffen (oder genauer: von dem Zielspeicher der Datenzugriffe) abhängig ist, ändert sie sich bei der Auslagerung eines Datenobjektes in den Scratchpad-Speicher.

Im ILP werden daher die maximalen Ausführungszeiten der Basisblöcke unter der Voraussetzung, dass sich alle Datenobjekte im Hauptspeicher befinden, als Konstanten $wcet_b$ definiert. Diese können durch eine statische Analyse des Programms vor der Optimierung bestimmt werden. Der Gewinn, welcher durch die Scratchpad-Allokation eines Objektes d für den Block b erzielt werden kann, wird als $gain_{b,d}$ bezeichnet. Er hängt unter anderem von der Anzahl und der Art der Zugriffe auf d innerhalb von b ab. Auf einigen Architekturen hat auch die Art oder Reihenfolge der Instruktionen innerhalb des Basisblockes einen Einfluss auf den möglichen Gewinn. Dieser Wert kann berechnet werden (wenn entsprechende Informationen über die Zielarchitektur bekannt sind) oder ebenfalls durch eine statische Analyse ermittelt werden. Da letzteres jedoch zahlreiche Analysevorgänge erfordern würde, sollte versucht werden, den Wert rechnerisch zu ermitteln. Selbstverständlich wird nicht in jedem Basisblock auf alle möglichen Datenobjekte zugegriffen. Für diesen Fall gilt einfach $gain_{b,d} = 0$.

Sind diese Eingabedaten bekannt, kann für jeden Basisblock eine Variable C_b eingeführt werden, welche dessen Kosten als die WCET des Blockes abzüglich der Gewinne für alle ausgelagerten Objekte definiert:

$$\forall b \in basic\_blocks: \qquad C_b = wcet_b - \sum_{d \in data\_objects} S_d * gain_{b,d} \tag{4.4}$$

4.4 Modellierung des Kontrollflusses

Die Modellierung des Kontrollflusses soll zunächst für einen Programmabschnitt ohne Schleifen gezeigt werden, da diese eine besondere Behandlung erfordern, die später erläutert wird. O.B.d.A. wird im Folgenden davon ausgegangen, dass jeder Kontrollflussgraph einen eindeutigen Start- und Endknoten besitzt. Dies ist keine Einschränkung, da durch das Erzeugen zweier Dummyknoten mit den Kosten 0 und dem Hinzufügen der entsprechenden Kanten eine solche Konstellation aus beliebigen Graphen leicht erzeugt werden kann.

Es sollen nun für die Kanten dieses Graphen Constraints erzeugt werden, so dass der vorhandene Kontrollfluss im ILP abgebildet wird. Dazu wird zunächst für jeden Knoten n eine weitere Variable W_n benötigt, die dessen Gewicht repräsentiert. An dieser Stelle ist zu beachten, dass im vorherigen Abschnitt die Kosten eines Basisblocks definiert wurden, nun jedoch vom Gewicht

eines Knotens die Rede ist. Auf unterster Ebene steht jeder Knoten noch für genau einen Basisblock. Im Folgenden können jedoch auch mehrere Basisblöcke zu einem Knoten zusammengefasst werden, so dass diese Äquivalenz dann nicht mehr gilt.

Für den Endknoten des Kontrollflussgraphen wird das Gewicht auf die Kosten des zugehörigen Basisblockes gesetzt:

$$W_{sink} = C_{sink} \tag{4.5}$$

Für jede Kante $a \rightarrow b$ wird nun ein Constraint erzeugt, der das Gewicht des Knotens a mindestens auf das Gewicht des Nachfolgeknotens (W_b) zuzüglich der Kosten im Knoten selbst (C_a) setzt.

$$W_a \geq W_b + C_a \tag{4.6}$$

Durch diese Formulierung wachsen die Knotengewichte startend beim Endknoten entgegen dem Verlauf des Kontrollflusses. Im Startknoten entspricht das Gewicht dann einer WCET-Abschätzung des gesamten Graphen. Hierbei wird implizit der Worst-Case Execution Pfad berücksichtigt, da beim Zusammenlaufen mehrerer Kanten in einem Knoten immer das höchste Knotengewicht ausschlaggebend für das Gewicht des nächsten Knotens ist.

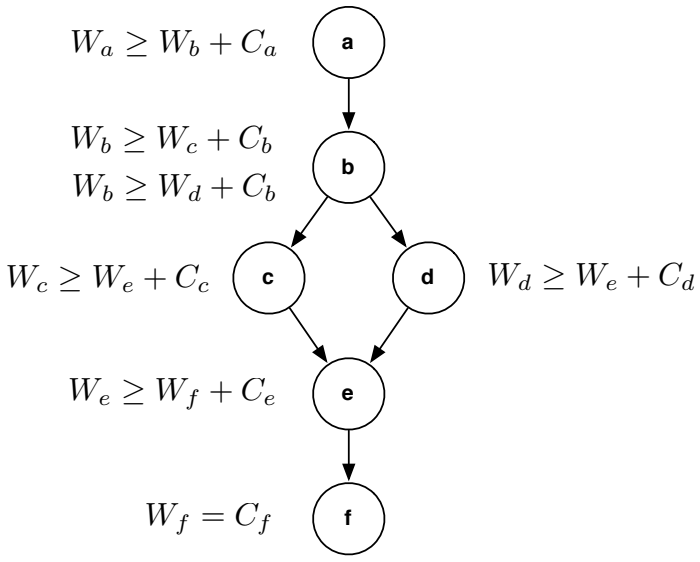

Abbildung 4.1: Constraints für einen Kontrollflussgraphen ohne Schleifen

In Abbildung 4.1 sind die erzeugten Constraints für einen simplen Kontrollflussgraphen dargestellt. Hier gibt es zwei mögliche Worst-Case Execution Pfade: $a \rightarrow b \rightarrow c \rightarrow e \rightarrow f$ oder $a \rightarrow b \rightarrow d \rightarrow e \rightarrow f$. Da für beide ausgehenden Kanten von b eine Restriktion erzeugt wird, muss das Gewicht von b immer über dem von c und dem von d (zuzüglich der Kosten von b selbst) liegen. Damit wird in jedem Fall der ungünstigere der beiden Pfade berücksichtigt.

4.4.1 Behandlung von Schleifen

Die Verwendung von Schleifen im Programm führt zu Zyklen im Kontrollflussgraphen. Diese können nicht durch die beschriebenen Constraints abgebildet werden, da dies das Gewicht eines Knotens innerhalb einer Schleife ins Unendliche steigen lassen würde. Zudem muss die Anzahl der Schleifeniterationen berücksichtigt werden. Zu diesem Zweck ist es zunächst nötig, die Schleifeniterationsgrenzen zu kennen. Die maximale Ausführungsanzahl einer Schleife, die im Basisblock b beginnt, ist eine Eingabe des Algorithmus und soll als $wcec_b$ bezeichnet werden. Der entsprechende Wert kann durch eine automatische Schleifenanalyse oder durch manuelle Annotation im Quellcode bereitgestellt werden.

Damit nun auch zyklische Kontrollflussgraphen behandelt werden können, werden zunächst die Knoten der innersten Schleifen des Programms betrachtet. Diese besitzen bis auf die Rückkante(n) der Schleife keine weiteren zyklischen Kanten. Daher können für alle Kanten des Schleifenkörpers wie oben beschrieben Constraints erzeugt werden. Das Gewicht des Eintrittsknotens der Schleife entspricht dann einer WCET-Abschätzung eines Schleifendurchlaufs. Multipliziert man dieses Gewicht mit der Anzahl der Schleifeniterationen, erhält man eine Abschätzung der WCET für die gesamte Schleife. Diese kann dann im Folgenden als eigener Knoten des Kontrollflussgraphen behandelt werden, der alle Knoten innerhalb der Schleife ersetzt. Die Kosten dieses Knotens haben nun keinen direkten Zusammenhang mehr zu den Kosten eines bestimmten Basisblockes. Sie werden durch einen Constraint folgendermaßen definiert:

$$C_{loop} = W_{first\_node(loop)} * wcec_{first\_node(loop)} \tag{4.7}$$

Da für den neu entstandenen Knoten nun die Kosten definiert sind, kann er im Folgenden genau wie alle weiteren Knoten behandelt werden.

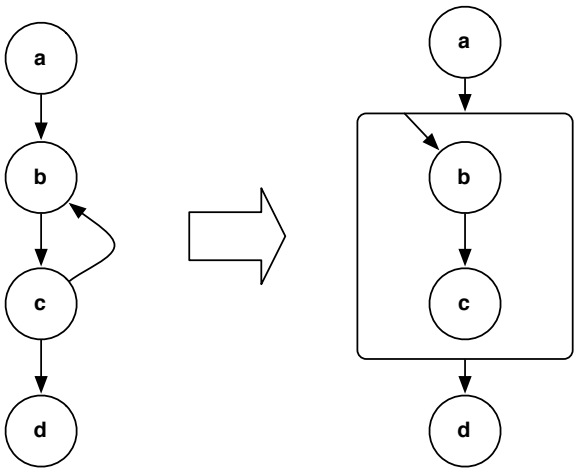

Abbildung 4.2: Behandlung von Schleifen

Abbildung 4.2 verdeutlicht dieses Vorgehen anhand eines Beispiels: Die Schleife mit den Knoten b und c wird durch einen neuen Knoten ersetzt, der den Schleifenkörper ohne die Rückkante repräsentiert. Während dieses Vorgangs wird wie oben beschrieben ein Constraint für die Kante $b \rightarrow c$ erzeugt. Die Knoten a und d erhalten jetzt als neuen Vorgänger bzw. Nachfolger den

erzeugten Schleifenknoten.

Wie bereits angedeutet, kann die Behandlung nun auf der nächsten Ebene von Schleifen fortgeführt werden. So wird, beginnend bei den innersten Schleifen, der gesamte Kontrollflussgraph behandelt, bis alle Zyklen beseitigt sind. Werden nun für die verbliebenen Kanten die entsprechenden Constraints erzeugt, entspricht das Gewicht des ersten Knotens der WCET-Abschätzung des gesamten Graphen.

Diese iterative Kontraktion von Schleifen ist in Abbildung 4.3 dargestellt. Zunächst wird die innerste Schleife mit den Knoten c und d behandelt. Durch die Kontraktion dieser beiden Knoten entsteht der neue Knoten *c_loop*. Dieser ist wiederum Teil eines Schleifenkörpers und wird somit im nächsten Schritt mit b, e und f zu *b_loop* zusammengefasst. Nun sind alle Schleifen beseitigt und die Gewichte der drei übrigen Knoten können durch Constraints definiert werden.

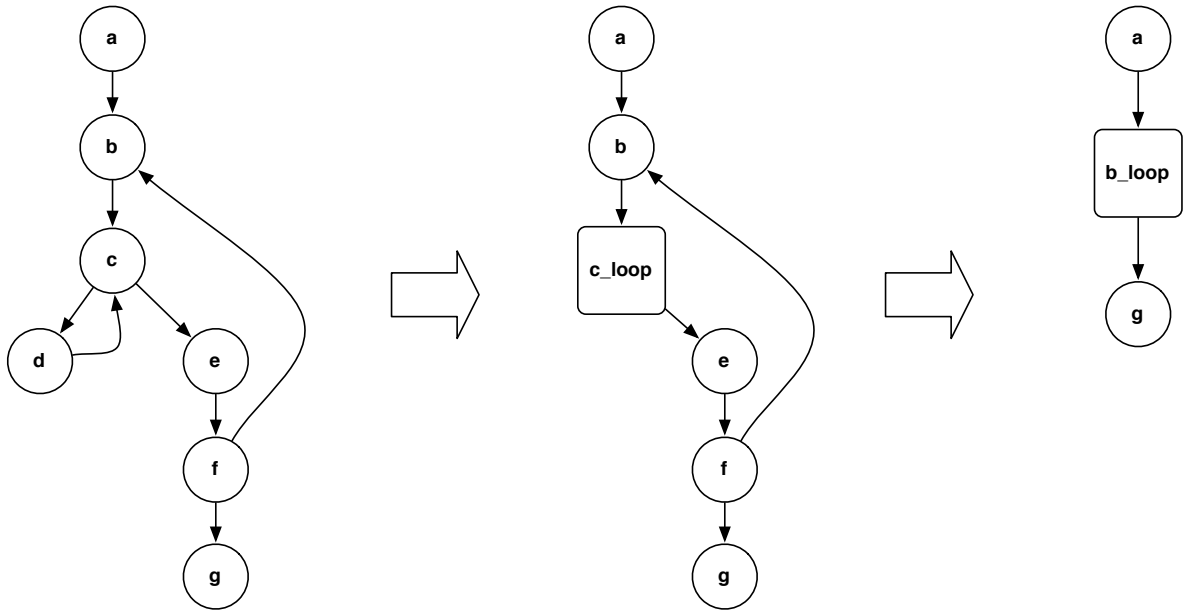

Abbildung 4.3: Iterative Kontraktion von Schleifen

4.4.2 Behandlung von Funktionsaufrufen

Bisher wurden noch keine Funktionsaufrufe behandelt, da diese nicht durch Kanten im Kontrollflussgraphen dargestellt werden. Dies ist nicht möglich, da eine Funktion potentiell von unterschiedlichen Punkten angesprungen werden kann und somit die Return-Kanten beim Verlassen der Funktion nicht eindeutig wären. In Abbildung 4.4 rufen sowohl a als auch b die Funktion *func1* auf. In y ist daher nicht klar, zu welchem dieser Blöcke der Rücksprung erfolgen soll. Würde man hier beide Möglichkeiten als Kanten einfügen, entstünde ein Zyklus im Kontrollflussgraphen, der die Berechnung der Knotengewichte unmöglich machen würde.

Die Kosten eines Funktionsaufrufs müssen daher direkt zu dem Gewicht des aufrufenden Knotens addiert werden. Dazu wird zunächst jede Funktion einzeln behandelt und die entsprechenden Knotengewichte durch Constraints festgelegt. Das Gewicht des Startknotens einer Funktion

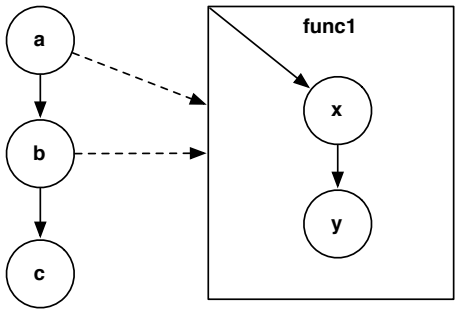

Abbildung 4.4: Aufruf einer Funktion von unterschiedlichen Stellen im Programm

entspricht dann der WCET-Abschätzung der gesamten Funktion. Somit können die Kosten einer Funktion definiert werden als:

$$C_{func} = W_{first\_node(func)} \tag{4.8}$$

Diese Kosten müssen dann in jedem Basisblock berücksichtigt werden, der die entsprechende Funktion aufruft. Da jeder Basisblock nur genau eine Funktion aufrufen kann (und zwar nur durch seine letzte Instruktion), kann Constraint 4.4 für Blöcke, die einen Funktionsaufruf beinhalten, wie folgt erweitert werden:

$$\begin{aligned} &\forall b \in basic\_blocks: \\ &C_b = wcet_b + C_{called\_func(b)} - \sum_{d \in data\_objects} S_d * gain_{b,d} \end{aligned} \tag{4.9}$$

Jedes Programm besitzt genau eine Startfunktion. Meist ist dies die Funktion *main*, daher soll dieser Name auch hier dafür verwendet werden.

Behandlung rekursiver Funktionsaufrufe

Es verbleibt noch eine Situation, die bisher nicht behandelt werden kann: Rekursive Funktionsaufrufe. Zur Beschreibung dieser soll zunächst der Begriff des *Call-Graphen* eingeführt werden.

Ein Call-Graph ist ein Graph, dessen Knoten die Funktionen eines Programms repräsentieren. Wird innerhalb einer Funktion $f1$ eine andere Funktion $f2$ aufgerufen, so wird dies durch eine Kante $f1 \rightarrow f2$ im Call-Graphen dargestellt. Abbildung 4.5 zeigt einen Call-Graphen mit zwei unterschiedlichen Arten von Rekursion:

Die Funktion $f1$ ruft sich selbst auf. Dies wird als *direkte Rekursion* bezeichnet. In $f3$ ist die Situation ein wenig komplizierter. Diese Funktion ruft eine andere Funktion ($f2$) auf, von der aus jedoch $f3$ im Call-Graphen ebenfalls erreichbar ist. Dies nennt man *indirekte Rekursion.*

Damit für solche Programme überhaupt eine statische WCET-Analyse möglich ist, ist es nötig, die maximale Rekursionstiefe für alle Aufrufe rekursiver Funktionen zu kennen. Diese Informationen können dann auch im ILP genutzt werden, um solche Funktionen zu behandeln. Direkte

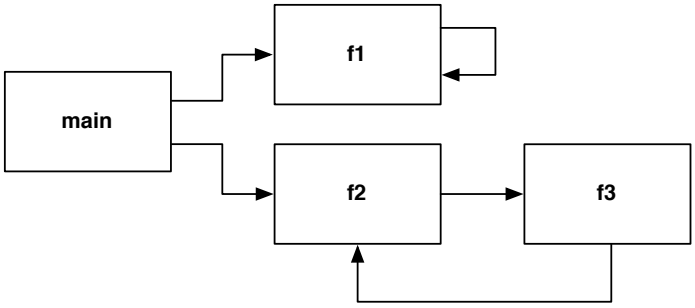

Abbildung 4.5: Rekursion im Call-Graphen

Rekursion lässt sich damit durch eine zusätzliche Erweiterung der Blockkosten-Constraints aller die Funktion aufrufenden Basisblöcke abbilden:

$$\forall b \in basic\_blocks:$$
$$C_b = wcet_b + recursion\_depth(b) * C_{called\_func(b)} - \sum_{d \in data\_objects} S_d * gain_{b,d} \tag{4.10}$$

Für die Basisblöcke innerhalb der Funktion selbst dürfen die Kosten des rekursiven Aufrufs dann natürlich nicht mehr hinzugerechnet werden, diese werden ja durch die Multiplikation der Funktionskosten mit der Rekursionstiefe bereits berücksichtigt. Für indirekte Rekursionen ist die Anwendung dieses Verfahrens leider nicht möglich. Eine Erklärung dafür wird im Abschnitt 4.6 gegeben.

4.5 Zielfunktion und Auswertung

Ist der Kontrollfluss des Programms komplett durch Restriktionen abgebildet, so entspricht das Gewicht des ersten Knotens in *main* der WCET-Abschätzung des Programms unter Berücksichtigung der durch die Entscheidungsvariablen S_d bestimmten Scratchpad-Belegung.

Das Ziel des Algorithmus ist die Minimierung der Worst-Case Execution Time des Programms, daher muss durch die Zielfunktion des ILP das Gewicht des Startknotens minimiert werden:

$$W_{first\_node(main)} \longrightarrow min \tag{4.11}$$

Durch Constraint 4.8 kann dies auch als Minimierung der Kosten der Startfunktion ausgedrückt werden:

$$C_{main} \longrightarrow min \tag{4.12}$$

Das ILP wird so unter Berücksichtigung der Scratchpad-Kapazität eine optimale Belegung der Entscheidungsvariablen bestimmen, da durch die Auslagerung eines Datenobjektes wie in Abschnitt 4.3 beschrieben die Kosten aller auf dieses Objekt zugreifenden Basisblöcke gesenkt

werden. Dies verringert auch die Kosten aller Pfade, die diesen Block enthalten, was sich in geringeren Knotengewichten widerspiegelt. Somit wird implizit eine Belegung gewählt, welche die Ausführungszeit des ungünstigsten Pfades minimiert. Eine explizite Behandlung des Worst-Case Execution Pfades ist daher in dieser ILP-Formulierung nicht nötig.

Nach der Lösung des ILP können die Datenobjekte d, für die $S_d = 1$ gilt, in den Scratchpad-Speicher ausgelagert werden.

Es ist anzumerken, dass die so berechnete WCET des Programms (also das Gewicht des Startknotens) nicht der Abschätzung durch eine statische Analyse entsprechen muss. Sie stellt also nicht zwangsläufig eine sichere WCET-Abschätzung dar und sollte auch nicht als solche verwendet werden. Dies hat unterschiedliche Ursachen, die im nächsten Abschnitt behandelt werden. Sie hat jedoch genug Aussagekraft, um als zu minimierender Wert in der Zielfunktion zu dienen.

4.6 Einschränkungen

Obwohl der vorgestellte Algorithmus sehr vielseitig einsetzbar ist, gibt es eine Reihe von Beschränkungen, die bei der Anwendung beachtet werden müssen. Diese machen eine Nutzung des Algorithmus nicht unmöglich, verringern aber unter Umständen die Qualität der Ergebnisse. Lässt man die hier aufgeführten Punkte außer Acht, so berechnet das ILP unter der Voraussetzung der Korrektheit aller Eingabedaten eine optimale Lösung.

Nebeneffekte auf der Zielhardware

Leider kann nicht ausgeschlossen werden, dass bei Ausführung eines optimierten Programms auf der Zielhardware Nebeneffekte eintreten. Bei der Berechnung der Blockkosten wird zum Beispiel der Wert $gain_{b,d}$ subtrahiert, der dem Gewinn entspricht, den eine Scratchpad-Auslagerung von d im Block b erzielen würde. Auf manchen Plattformen kann jedoch durch den veränderten Zustand der Pipeline die Auslagerung eines Objektes in einem Block auch Einfluss auf die Ausführungszeiten der darauf folgenden Blöcke haben. Dies kann leider im ILP nicht berücksichtigt werden.

Unausführbare Pfade im Kontrollflussgraphen

Es kann vorkommen, dass bestimmte Pfade im Kontrollflussgraphen niemals ausgeführt werden können. Solche Pfade heißen *unausführbar (infeasible)*. Ein Beispiel für einen unausführbaren Pfad enthält das Programm in Abbildung 4.6.

Hier wird der Code in Block B genau dann ausgeführt, wenn die Variable x den Wert 0 hat. Dann wird der Wert von y auf 1 gesetzt, so dass die Bedingung in Block C niemals als wahr ausgewertet wird. Der Pfad $A \rightarrow B \rightarrow C \rightarrow D$ ist somit unausführbar.

Eine solche Beschränkung der möglichen Pfade wird innerhalb des ILP nicht berücksichtigt. Daher würde im schlimmsten Fall entlang eines Worst-Case Pfades optimiert, der gar nicht ausführbar ist. Dies lässt sich leider nicht verhindern, da hierzu die Menge der unausführbaren Pfade eine Eingabe des Algorithmus darstellen müsste. Diese Information wird jedoch zumindest von dem in dieser Arbeit verwendeten WCET-Analyse Tool aiT nicht bereitgestellt.

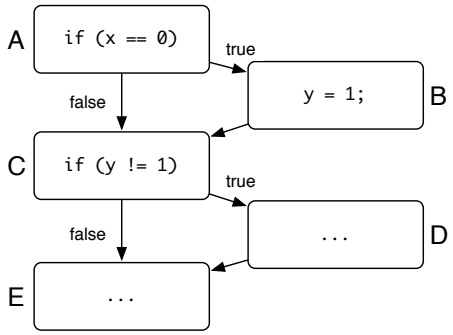

Abbildung 4.6: Unausführbarer Pfad im Kontrollflussgraphen

Kanten, die in keinem einzigen ausführbaren Pfad enthalten sind, können jedoch behandelt werden: Für diese wird einfach keine entsprechende Restriktion erzeugt.

Schleifen mit mehreren Eintrittspunkten bzw. Austrittspunkten

Besitzt eine Schleife mehrere Eintritts- oder Austrittspunkte, so kann der erste bzw. letzte Schleifendurchlauf eine andere Ausführungszeit besitzen als die übrigen Durchläufe. Der beschriebene Algorithmus wird auch für diese Fälle die WCET des Schleifenrumpfes nutzen, so dass die berechnete gesamte Laufzeit aller Schleifeniterationen etwas höher als der tatsächliche Wert ist.

Unterschiedliche Eintrittspunkte von Schleifen sind in strukturierten Programmen sehr selten und treten in C-Programmen hauptsächlich durch Verwendung des `goto` Befehls auf. Die Betrachtung dieser kann daher vernachlässigt werden. Verschiedene Austrittspunkte hingegen können auch durch den Aufruf von `break` innerhalb einer Schleife erzeugt werden. Leider liegen während der Erzeugung des ILP keine Informationen darüber vor, ob und wann ein solcher Befehl tatsächlich ausgeführt wird, daher kann dieser Fall nicht behandelt werden.

Da jedoch Schleifen im Allgemeinen recht häufig ausgeführt werden und an dieser Stelle ohnehin nur die maximale Ausführungszeit aller Schleifeniterationen betrachtet wird, ist die Wahrscheinlichkeit einer Verschlechterung des Ergebnisses durch diese Einschränkungen sehr gering.

Indirekte Rekursion

Die Behandlung von direkter Rekursion wurde im Abschnitt 4.4.2 beschrieben. Für Programme, in denen indirekte rekursive Aufrufe auftreten, existiert jedoch ein Problem, das eine solche Behandlung unmöglich macht. Dies soll anhand des Beispiels in Abbildung 4.7 gezeigt werden.

In diesem Beispiel kann nicht bestimmt werden, an welcher Stelle die indirekte Rekursion zwischen $f1$ und $f2$ behandelt werden muss, da beide Funktionen direkt von *main* aufgerufen werden. Für eine Auflösung der Rekursion müsste die maximale Rekursionstiefe mit den Kosten der entsprechenden Funktion multipliziert werden und dann alle Kanten entfernt werden, die einen rekursiven Aufruf darstellen. Würde der Aufruf von $f1$ an Kante a behandelt, müsste also die Kante c entfernt werden. Wenn jedoch der Aufruf von $f2$ an Kante d behandelt wird, so müsste die Kante b entfernt werden.

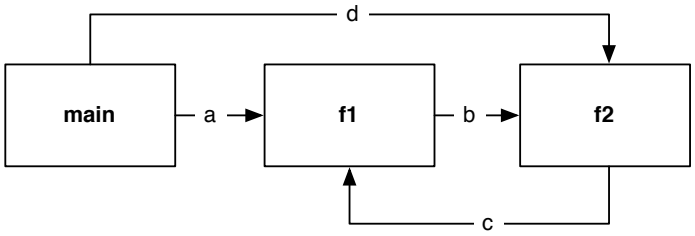

Abbildung 4.7: Indirekte Rekursion

Durch diesen Widerspruch ist eine Auflösung der Rekursion nicht möglich. Eine Behandlung ähnlich der von Schleifen wäre zwar möglich, die so entstehenden Werte wären jedoch wenig aussagekräftig, da so bei tiefer verschachtelten indirekten Funktionsaufrufen eventuell die Kosten von Funktionen hinzugerechnet würden, die gar nicht aufgerufen werden. In praxisrelevanten Programmen, deren WCET-Abschätzung tatsächlich wichtig ist, würde man ohnehin versuchen, solche Konstrukte zu vermeiden, da diese auch für die statische WCET-Analyse eine potenzielle Quelle großer Überschätzungen darstellen.

Kapitel 5

Umsetzung

Dieses Kapitel beschreibt die Umsetzung der WCET-Optimierung durch die statische Scratchpad-Allokation von Daten innerhalb des WCC. Neben der Implementierung des eigentlichen Algorithmus waren zahlreiche Erweiterungen oder Anpassungen anderer Compilerkomponenten notwendig, um die Eingabedaten bereitzustellen und die Auslagerung durchzuführen.

Im ersten Abschnitt dieses Kapitels soll zunächst ein grober Überblick über die Integration der Optimierung in den Compiler gegeben werden. Im Anschluss werden die einzelnen Teilschritte, die für deren Umsetzung nötig waren, detaillierter beschrieben: Zunächst wird erläutert, wie Informationen über die Datenzugriffe einzelner Instruktionen gewonnen werden. Daraufhin werden die Datenstrukturen beschrieben, mit denen der Algorithmus arbeitet. Schließlich wird der Aufbau des ILP und die Belegung der darin verwendeten plattformspezifischen Konstanten erläutert.

5.1 Integration in den WCC

Um eine Scratchpad-Allokation von Daten auf Basis von WCET-Informationen in den Compiler zu integrieren, waren Erweiterungen und Anpassungen an mehreren Stellen nötig. Neben der Implementierung des eigentlichen ILP-Algorithmus müssen dessen Eingabedaten bereitgestellt werden und die Ergebnisse umgesetzt werden, so dass die Datenobjekte in der erzeugten Binärdatei tatsächlich in den Scratchpad-Speicher ausgelagert werden. In Kapitel 3.3 wurde der Aufbau des WCC vor der Integration der Scratchpad-Allokation beschrieben. In diesem Abschnitt soll nun ein grober Überblick darüber gegeben werden, welche Komponenten geändert und hinzugefügt werden mussten.

In Abbildung 5.1 sind alle Komponenten und Zwischendarstellungen, die im Verlauf dieser Arbeit modifiziert wurden, grau hinterlegt. Es ist zu sehen, dass das Compiler-Frontend (Parser und High-Level Optimierungen) von den Änderungen nicht betroffen ist. Eine Änderung dieser Komponenten war nicht notwendig, da für die Nutzung der Optimierung keinerlei Modifikation des zu übersetzenden Quelltextes notwendig ist. Sie lässt sich also ohne weitere Nutzerinteraktion für beliebige mit dem WCC übersetzbare Programme aktivieren. Dies geschieht über die Angabe eines Kommandozeilenflags beim Aufruf des Compilers.

Es soll nun kurz dargestellt werden, welche Änderungen in den einzelnen Komponenten des

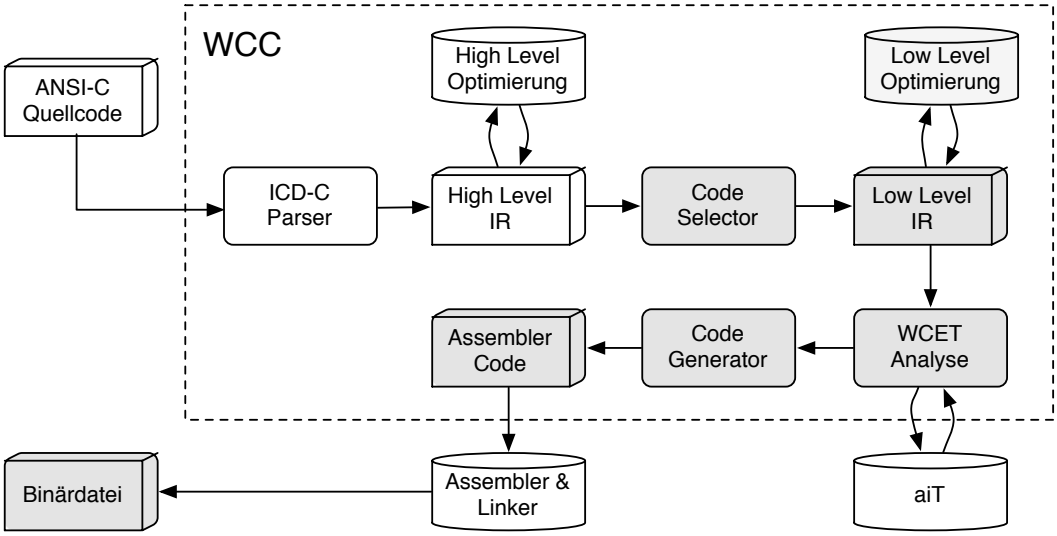

Abbildung 5.1: Geänderte Komponenten des WCC

WCC vorgenommen wurden. Die Umsetzung dieser Änderungen und weitere Details werden in den folgenden Abschnitten dieses Kapitels genauer betrachtet.

- **Code Selector**
  Im Code Selector wird die High-Level Zwischendarstellung in eine Low-Level Zwischendarstellung übersetzt. An dieser Stelle werden erstmals architekturspezifische Details behandelt, es gehen allerdings einige Informationen über die Semantik des zu übersetzenden Programms verloren. Dazu gehören auch Informationen darüber, welche Variablen die Anweisungen des C-Programms referenzieren. Es muss hier also dafür gesorgt werden, dass auch auf LLIR-Ebene noch nachvollzogen werden kann, auf welche Datenobjekte einzelne Instruktionen zugreifen. Die Änderungen am Code Selector werden im Abschnitt 5.2 behandelt.

- **WCET Analyse**
  Vor Beginn dieser Arbeit waren in der WCET-Analyse keine Informationen über den Speicherort von Daten verfügbar. Daher mussten die Konverter, welche die LLIR-Darstellung in das aiT-Eingabeformat CRL und zurück übersetzen, entsprechend erweitert werden. Zusätzlich können die Ergebnisse der aiT-Analyse genutzt werden, um die im Code Selector erstellten Datenzugriffsinformationen zu verfeinern und zu ergänzen. Auch diese Änderungen sind im Abschnitt 5.2 beschrieben.

- **Low Level Optimierungen**
  Da es sich bei der Scratchpad-Allokation um eine Optimierung auf LLIR-Ebene handelt, ist die Implementierung des eigentlichen Algorithmus in dieser Komponente untergebracht. Hier wird für die einzelnen Datenobjekte der optimale Speicherort gewählt und diese Information in der LLIR gespeichert. Zudem wird, sofern die Optimierung aktiviert ist, die WCET-Analyse nun an dieser Stelle aufgerufen. Dies geschieht vor und nach der Bestimmung der Speicherorte, um die durch die Allokation erzielte Verbesserung bewerten zu können. Genauer wird der Ablauf im Abschnitt 5.1.1 beschrieben.

- **Code Generator**
  Der Code Generator erzeugt aus der LLIR den Assemblercode, aus dem später der ausführbare Maschinencode generiert wird. An dieser Stelle müssen die Speicherorte der Datenobjekte berücksichtigt werden, um diese tatsächlich in den entsprechenden Speicherbereich einzulagern. Diese recht triviale Erweiterung wird kurz am Ende des Abschnitts 5.1.2 behandelt.

Abbildung 5.2 zeigt den Ablauf aller für diese Arbeit relevanten Schritte etwas detaillierter. Insbesondere lässt sich hier erkennen, dass die Scratchpad-Allokation nach allen weiteren LLIR-Optimierungen durchgeführt werden muss. Dies ist notwendig, da jede andere Optimierung beliebige Codetransformationen vornehmen kann. Die Bestimmung des Speicherortes von Datenobjekten ist jedoch nur dann sinnvoll, wenn der Programmablauf, und damit die Anzahl und Position von Datenzugriffen, bereits feststeht.

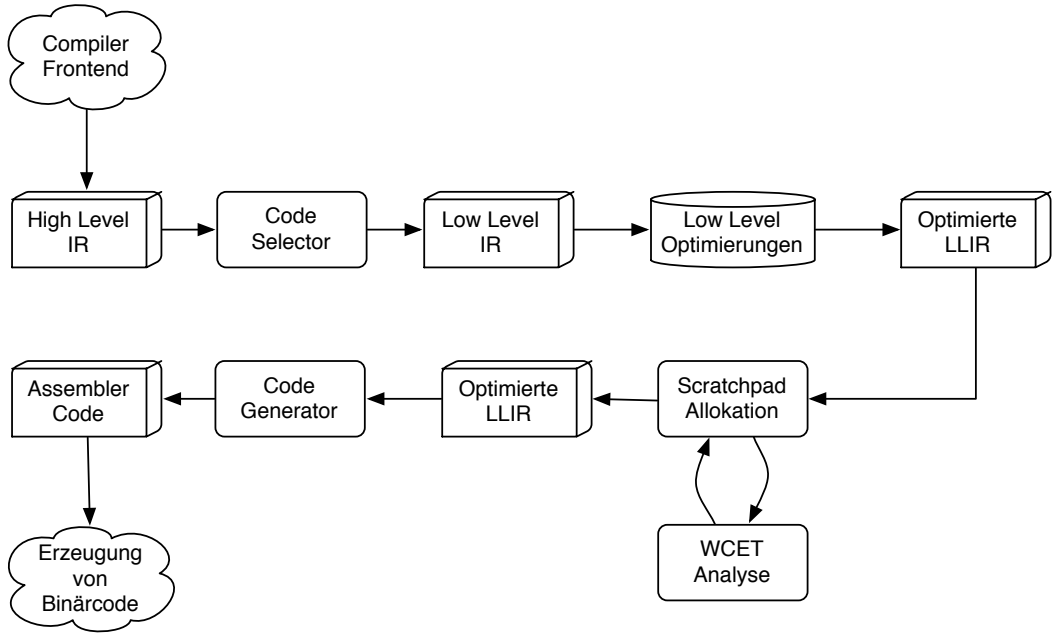

Abbildung 5.2: Integration der Optimierung in den WCC

5.1.1 Ablauf der Optimierung

Die Implementierung der Scratchpad-Allokation selbst besteht ebenfalls aus einer Reihe unterschiedlicher Schritte (s. Abbildung 5.3).

1. WCET-Analyse
   Am Anfang der Optimierung liegt das zu optimierende Programm in Form von einer oder mehreren LLIR-Instanzen vor. Alle weiteren Optimierungen wurden bereits vorgenommen, so dass der Programmablauf nicht mehr geändert wird. Zudem enthalten die einzelnen Instruktionen Informationen darüber, auf welche Datenobjekte sie zugreifen. Die

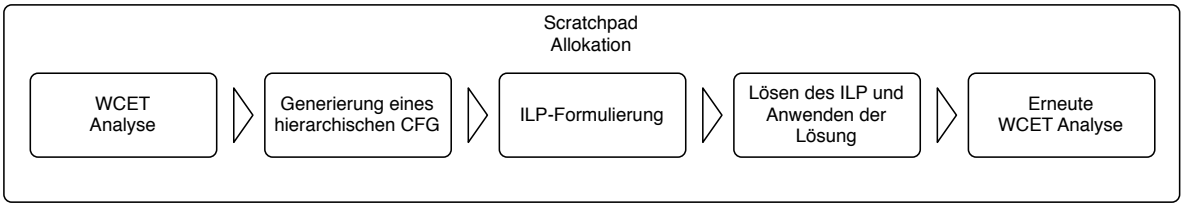

Abbildung 5.3: Ablauf der Scratchpad-Allokation

Datenobjekte selbst befinden sich jedoch allesamt im SRAM. Die vorliegende Zwischenrepräsentation entspricht also genau dem Programm, das ohne eine Scratchpad-Allokation erzeugt werden würde.

So können durch einen Aufruf der WCET-Analyse an dieser Stelle die WCET-Daten der einzelnen Basisblöcke ohne Nutzung des Scratchpad-Speichers bestimmt werden. Auch eine Reihe von weiteren durch aiT bereitgestellten Informationen, die später benötigt werden, sind daraufhin verfügbar. Dazu gehört beispielsweise die Kennzeichnung unausführbarer Kanten und Blöcke, die im weiteren Verlauf nicht mehr berücksichtigt werden müssen.

Nach einer erfolgreichen WCET-Analyse stehen diese Daten innerhalb der LLIR zur Verfügung, so dass alle folgenden Schritte darauf zugreifen können.

2. Generierung eines verschachtelten Kontrollflussgraphen
   An sich stellt bereits die LLIR genügend Informationen bereit, um die einzelnen Basisblöcke als Knoten eines Kontrollflussgraphen zu behandeln. Um jedoch den Kontrollfluss in Form von Constraints innerhalb der ILP-Formulierung des Optimierungsproblems zu modellieren, sind einige Transformationen erforderlich. Insbesondere das in Kapitel 4.4 beschriebene Erzeugen von einzelnen Knoten für ganze Schleifen lässt sich alleine mit LLIR-Elementen nicht beschreiben. Daher wird die Low-Level Repräsentation des Programms in eine Darstellung als verschachtelter Kontrollflussgraph überführt.

   Eine Beschreibung der Datenstruktur zur Darstellung des Kontrollflussgraphen und des Algorithmus zu dessen Aufbau findet sich in Abschnitt 5.3.

3. Formulierung des ILP
   Aus dem im vorherigen Schritt erstellten Kontrollflussgraphen kann nun eine Formulierung als ILP erstellt werden. Dazu werden neben dem CFG selbst auch eine Reihe weiterer Informationen, beispielsweise Speicherzugriffszeiten und -größen und die Datenzugriffsinformationen einzelner Instruktionen benötigt. Insbesondere die Berechnung des durch Auslagerung eines Datenobjektes erzeugten Gewinns pro Block stellt hier eine Herausforderung dar. All diese Details werden in Abschnitt 5.4 behandelt.

4. Lösen des ILP und Anwenden der Lösung
   Die Lösung des ganzzahligen linearen Optimierungsproblems geschieht durch einen externen LP-Solver. Nachdem so für jedes Datenobjekt der Speicherort bestimmt wurde, müssen diese Informationen innerhalb der LLIR angewendet werden. Nach Durchführung dieses Schrittes sind somit die endgültigen Speicherorte aller Datenobjekte festgelegt.

5. Erneute WCET-Analyse
   In einem zweiten Durchlauf der aiT Analyse kann nun eine endgültige WCET-Abschätzung

für das Programm gewonnen werden. Zudem kann durch Vergleich mit den Daten des ersten Analysevorgangs das Ergebnis der Optimierung bewertet werden. Dies ist der Grund, weshalb auch die zweite WCET-Analyse direkt aus der Komponente zur Scratchpad-Allokation aufgerufen wird - in späteren Schritten des Compilers wären die ursprünglichen WCET-Informationen nicht mehr verfügbar, weil diese bei jedem Analysevorgang überschrieben werden.

Neben den hier gelisteten Änderungen und Erweiterungen von WCC-Komponenten waren noch eine Reihe von Anpassungen an der Infrastruktur des Compilers notwendig, die sich nicht in den Schaubildern darstellen lassen. Dazu gehört das Bereitstellen von Informationen über Speicherbereiche und die Zugehörigkeit von Datenobjekten zu diesen. Die zu diesem Zweck erfolgten Modifikationen des WCC werden im folgenden Abschnitt behandelt.

5.1.2 Repräsentation unterschiedlicher Speicherbereiche im WCC

Zu Beginn dieser Arbeit bestand innerhalb des WCC nicht die Möglichkeit, den Zielspeicher einzelner LLIR-Elemente zu bestimmen. Die Zuweisung zu bestimmten Speicherbereichen war fest vorgegeben und wurde beim Erzeugen des Assemblercodes vorgenommen. Somit wäre es nicht ohne weiteres möglich gewesen, einige Datenobjekte im Scratchpad und andere im SRAM zu speichern. Auch für Funktionen oder Basisblöcke war ein solcher Mechanismus nicht vorhanden. Zudem standen keine Informationen über Größe, Adressen oder Geschwindigkeit der verschiedenen Speicher der Zielarchitektur zur Verfügung.

Da für die in dieser Arbeit behandelte Scratchpad-Allokation verschiedene Komponenten des Compilers Zugriff auf diese Daten benötigen, war eine feste Kodierung innerhalb des Algorithmus von vornherein ausgeschlossen. Für eine Erweiterung der im WCC genutzten Datenstrukturen mussten allerdings einige Punkte beachtet werden. So ist die LLIR eine unabhängige Repräsentation, die auch in zahlreichen anderen Projekten genutzt wird. Eine Erweiterung der LLIR um WCC-spezifische oder sogar plattformabhängige Daten ist daher nicht möglich. Dennoch ist es wünschenswert, Informationen über den Speicherort von Elementen direkt auf dieser Ebene verfügbar zu machen, da so alle Komponenten im Compiler-Backend Zugriff darauf erhalten und bei Bedarf Änderungen vornehmen können.

Für eine in etwa zeitgleich mit dieser Arbeit gestartete Diplomarbeit, die sich mit der Auslagerung von Code in den Scratchpad Speicher beschäftigt, ergab sich das gleiche Problem. In Kooperation wurde daher ein Konzept entwickelt, die von den Arbeiten benötigten Informationen im WCC verfügbar zu machen. Die so entstandene Erweiterung ist in Abbildung 5.4 als Klassendiagramm dargestellt. Es ist zu erkennen, dass die LLIR selbst nur um zwei Klassen, *ObjectSectionLayout* und *ObjectSection*, erweitert wurde. Alle weiteren Änderungen betreffen lediglich den WCC. Durch die Erweiterung werden folgende Aufgaben erfüllt:

- **Zuordnung von LLIR-Elementen zu Sektionen**
  Die LLIR-Elemente zur Beschreibung von Basisblöcken und Datenobjekten benötigen eine direkte Verbindung zu dem Speicherbereich, in dem sie enthalten sind. Für Instruktionen ist dies nicht sinnvoll, da alle Instruktionen eines Basisblocks eine Einheit bilden, die sich nicht über verschiedene Speicherbereiche verteilen kann. Auch Funktionen kann eigentlich kein Speicherbereich zugewiesen werden, da diese im Assemblercode nicht mehr explizit

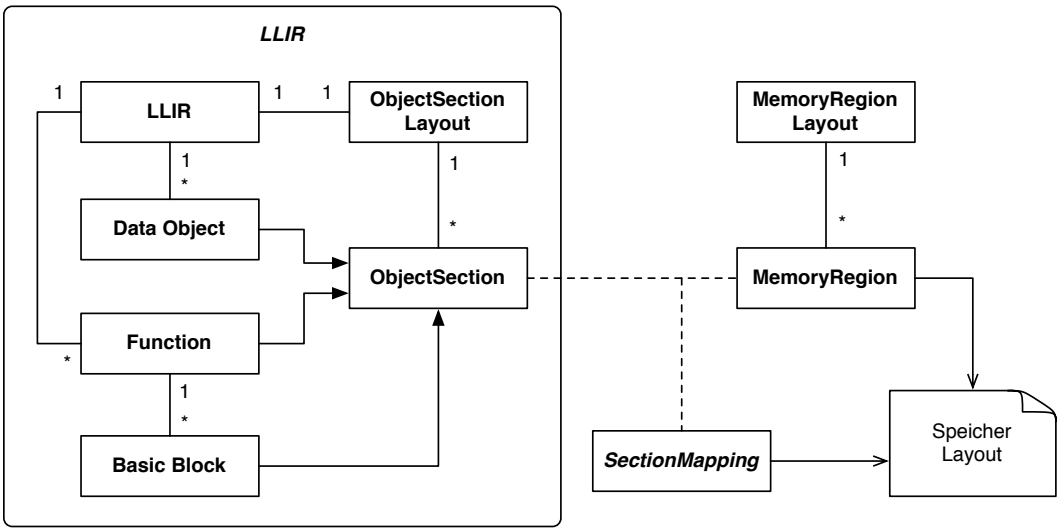

Abbildung 5.4: Speicherlayout im WCC

vorhanden sind. Dennoch kann die Zuweisung eines Speicherbereichs auch hier ermöglicht werden, um alle Basisblöcke der Funktion gleichzeitig zu verschieben.

Die so zugewiesenen Speicherbereiche dürfen allerdings keine plattformspezifischen Informationen enthalten, da dies die universelle Einsetzbarkeit der LLIR verhindern würde. Zu diesem Zweck wurden der LLIR so genannte *ObjectSections* hinzugefügt, die eigentlich nur einen Bezeichner ohne weitere Informationen darstellen. Jedem der drei oben genannten LLIR-Elemente kann ein solcher Bezeichner zugewiesen werden. Deren Verwaltung regelt die Komponente *ObjectSectionLayout*, von dem jede LLIR-Instanz genau eine enthält. Um die Eindeutigkeit von *ObjectSections* zu gewährleisten, geschieht das Erzeugen und Abrufen von diesen nur an dieser Stelle.

Eine *ObjectSection* entspricht im WCC jeweils einer Sektion der resultierenden Assemblerdatei, das heißt, deren Namen werden beim Schreiben des Assemblercodes für die `.section` Direktive genutzt. Zu jeder LLIR gehören daher automatisch die Standardsektionen `.text`, `.data` und `.bss`. Auch später hinzugefügte Sektionen werden auf diese Weise ausgegeben. Es ist also im Linkerskript des Compilers dafür zu sorgen, dass deren Namen dem Assembler bekannt sind.

- **Bereitstellen von Informationen über die Speicher der Zielarchitektur**
  Für hardwarenahe Optimierungen wie z.B. die Scratchpad-Allokation ist es notwendig, genaue Informationen über die Speicher der Zielarchitektur zur Verfügung zu stellen. Dazu gehören neben Größe und Adresse auch die Zugriffsrechte (Lesen, Schreiben, Ausführen) und die Zugriffsgeschwindigkeit, also die Anzahl der Wartezyklen beim Anfordern eines Datums aus dem Speicher. Letztere Information kann zwar unter Umständen deutlich variieren (z.B. abhängig von Pipeline- oder Buszustand), ermöglicht aber eine grobe Sortierung der Speicher nach deren Geschwindigkeit.

  All diese Daten können in einer Konfigurationsdatei des Compilers definiert werden, so dass eine Änderung leicht möglich ist. Auf diese Weise können auch neue Speicherbereiche definiert werden, zum Beispiel, wenn im Einsatzgebiet des zu übersetzenden Programms

externe Speicher vorhanden sind. Zudem lassen sich so auch hardwarespezifische Bereiche wie Stack oder CSA[1] mit diesem Mechanismus verschieben.

Die in der Speicherlayout-Konfigurationsdatei spezifizierten Informationen werden im WCC global durch das *MemoryRegionLayout* zur Verfügung gestellt. Hier lassen sich alle definierten Bereiche als Instanzen der *MemoryRegion* Klasse abrufen. Diese bietet dann Zugriff auf die Eigenschaften des entsprechenden Speicherbereichs.

- **Verknüpfung von Speicherbereichen und Sektionen**
  Die beiden bisher beschriebenen Punkte sind unabhängig voneinander. Es ist jedoch aus verschiedenen Gründen nötig, eine direkte Verknüpfung herzustellen. Um zum Beispiel eine WCET-Abschätzung unter Berücksichtigung der Speicherorte von LLIR-Elementen durch aiT zu ermöglichen, müssen deren absolute Speicheradressen bekannt sein. Auch während der Erzeugung von Assemblercode aus der LLIR werden Informationen über den zu einem Element gehörenden Speicherbereich benötigt.

  Es ist also eine Zuordnung von Sektionen zu Speicherbereichen nötig. Diese soll ebenfalls vom Nutzer definiert werden können und ist daher auch Bestandteil der Speicherlayout-Konfigurationsdatei. Das so erzeugte *SectionMapping* wird im WCC ebenfalls global zur Verfügung gestellt, so dass alle Komponenten Zugriff darauf haben.

  Während die Speicherregionen im WCC eindeutige und global abrufbare Objekte sind, wird die Verknüpfung zu Sektionen allerdings nur über Strings abgebildet. Es werden also nur die Namen der Sektionen mit Speicherbereichen verknüpft. Dies liegt zum einen daran, dass eine in der Konfigurationsdatei erwähnte Sektion nicht unbedingt innerhalb der LLIR tatsächlich existieren muss. So kann zum Beispiel die `.rodata` Sektion, welche zur Speicherung konstanter Daten genutzt wird, als zu einem bestimmten Speicherbereich zugehörig definiert sein, das zu übersetzende Programm aber gar keine konstanten globalen Daten enthalten. Eine entsprechende *ObjectSection* würde dann gar nicht existieren. Zudem sind die einzelnen LLIR-Instanzen im Compiler unabhängig voneinander, so dass sie jeweils eigene Instanzen der Sektionen besitzen.

Berücksichtigung der Speicherinformationen im Code Generator

Im Code Generator des WCC erfolgt die Übersetzung der Low-Level Zwischenrepräsentation in Asseblercode und die Speicherung in einer Datei, die später zu einer ausführbaren Binärdatei übersetzt wird. In dieser Assemblerdatei müssen allen Objekten die korrekten Sektionen zugewiesen werden. Dazu wurde die Auswertung der mit einem LLIR-Element verbundenen *ObjectSection* und das Schreiben dieser Sektionsinformationen in den Code Generator integriert. Eventuell notwenige Zusatzinformationen (beispielsweise über die möglichen Zugriffsarten auf eine Sektion) werden hierbei aus der über das *SectionMapping* verknüpften *MemoryRegion* gewonnen.

Da die tatsächliche Übersetzung in Maschinencode durch ein externes Tool, den GNU Linker, erfolgt, können die Informationen aus der WCC-spezifischen Speicherlayout-Konfigurationsdatei hier leider nicht genutzt werden. Der GNU Linker bezieht diese Daten aus einem Linkerskript, in dem ebenfalls die Verknüpfung von Speicherbereichen und Sektionen definiert ist. Der Anwender

[1] Context Save Area, ein Speicherbereich der TriCore-Architektur, auf dem die Rücksprungadresse und bestimmte Register bei Funktionsaufrufen automatisch gesichert werden

hat daher dafür Sorge zu tragen, dass Linkerskript und Konfigurationsdatei in diesen Punkten äquivalent sind. Für die Zukunft ist eine Erweiterung denkbar, welche das *SectionMapping* direkt aus dem Linkerskript erzeugt und somit redundante Angaben unnötig macht.

5.2 Erzeugung von Datenzugriffsinformationen

Für die Anwendung des in Kapitel 4 beschriebenen Algorithmus werden Informationen darüber benötigt, auf welche Datenobjekte innerhalb eines Basisblocks zugegriffen wird.

Viele Arbeiten zur Scratchpad-Allokation von Daten verwenden zu diesem Zweck ein Profiling-basiertes Verfahren (siehe z.B. [UB03, ABS02]). Dabei wird das zu untersuchende Programm in einem Simulator ausgeführt, der das Ziel der Speicherzugriffe dokumentiert. So können jedoch nur die Speicherzugriffe erkannt werden, welche für die während der Simulation genutzten Eingabedaten ausgeführt werden. Zudem erfordert diese Methode den Aufruf eines externen Tools, das auf bereits kompilierten Code arbeitet. Die Integration in den Workflow eines Compilers ist somit ausgeschlossen.

Daher musste ein Weg gefunden werden, die Datenzugriffsziele der LLIR-Instruktionen aus den im WCC verfügbaren Darstellungen des Programms zu berechnen. Zunächst war geplant, diese Daten komplett aus den Ergebnissen der statischen Analyse durch aiT zu gewinnen. Wie in Kapitel 3.2.3 beschrieben, integriert diese auch eine Wertanalyse, die mögliche Speicherziele für Load/Store Instruktionen berechnet. Leider stellte sich im Verlauf der Arbeit heraus, dass aiT nicht alle Speicherzugriffe hinreichend genau bestimmen kann. Besonders bei der Verwendung von Zeigern im Programmcode werden viele Zugriffe nicht erkannt. Daher musste eine geeignete Möglichkeit gefunden werden, auch komplexe Zugriffe zu erkennen.

5.2.1 Erkennung von Datenzugriffszielen

Da die LLIR eine Assembler-nahe Repräsentation ist, können aus ihr Informationen über Speicherzugriffe nur unter sehr großem Aufwand gewonnen werden. Auf dieser Ebene existieren bereits die Operationen, die für das Laden und Speichern von Daten aus dem bzw. in den Speicher verantwortlich sind. Leider ist in diesen Operationen aber kein direkter Bezug zu einer Variablen oder einer Speicheradresse vorhanden.

Der Adressraum der TriCore Architektur ist 32 Bit breit, die einzelnen Operationen unterstützen aber ebenfalls nur Breiten von 16 oder 32 Bit. Somit ist es nicht möglich, eine gesamte Adresse innerhalb eines Befehls zu kodieren. Es existieren zwar auch Load und Store Befehle mit direkter Adressierung, diese sind jedoch nur in Sonderfällen einsetzbar, da sie bestimmte Bits der Adresse implizit auf 0 setzen. Die meisten Speicherzugriffe nutzen daher eine (indizierte) Registeradressierung. Das heißt, dass zumindest ein Teil der endgültigen absoluten Adresse in einem Adressregister gespeichert ist. Ein vom Code Selector generierter Speicherzugriff wird meist in zwei Schritten durchgeführt: dem Laden der Adresse eines Datums in ein Adressregister und dem tatsächlichen Zugriff. Im Assemblercode sieht dies folgendermaßen aus:

```
movh.a %a12, HI:data
ld.w %d8, [%a12] LO:data
```

Hier werden im ersten Schritt die oberen 16 Bits der Adresse des Datums mit dem Bezeichner `data` in die oberen 16 Bits des Adressregisters `%a12` geladen. Die restlichen Bits des Registers werden dabei implizit auf 0 gesetzt. Erst mit dem zweiten Befehl werden die Daten aus dem Speicher in ein Datenregister geladen. Die tatsächlich zugegriffene Adresse berechnet sich dann aus der Summe des Inhalts des Adressregisters und des so genannten Offsets, das hier die unteren 16 Bit der Adresse des Datums darstellen. In diesem Beispiel ist ersichtlich, auf welches Datum zugegriffen wird, da dessen Label ein Operand des Load-Befehls ist. Dies ist aber nicht zwingend notwendig, denn das Label bezeichnet für den Assembler nur eine Adresse im Speicher und wird während der Übersetzung in Maschinencode durch genau diese ersetzt. Im Allgemeinen kann also bei isolierter Betrachtung eines Load/Store Befehls dessen Ziel nicht ermittelt werden. Es ist zusätzlich die Kenntnis über den Inhalt des Adressregisters notwendig.

Das Initialisieren des Adressregisters kann jedoch an einer viel früheren Stelle als der tatsächliche Datenzugriff erfolgen, und auch arithmetische Operationen darauf sind nicht ungewöhnlich, zum Beispiel beim Durchlaufen eines Arrays. Der Versuch, alle möglichen Werte eines Adressregisters für eine Load/Store Instruktion zu bestimmen, würde also eine Untersuchung der Auswirkungen aller Operationen, die eine Änderung des Registerinhalts zur Folge haben, erfordern. Im Prinzip ist dies das Vorgehen, das aiT in der Wertanalyse anwendet.

Es wird deutlich, dass die LLIR nicht genügend Informationen enthält, um aus dieser mit vertretbarem Aufwand die benötigten Informationen zu errechnen.

5.2.2 Datenzugriffsinformationen im Code Selector

Die im Compiler-Frontend genutzte Zwischenrepräsentation ICD-C ist eine dem C-Code des zu übersetzenden Programms sehr nahe Darstellung. Daher existiert hier noch ein direkter Zusammenhang zwischen Anweisungen und den darin referenzierten Variablen. Im WCC ist der Code Selector verantwortlich für die Übersetzung zwischen ICD-C und LLIR. Daher können an dieser Stelle die entsprechenden Informationen in die LLIR übertragen werden.

Da in der TriCore Architektur nur wenige Befehle auf den Speicher zugreifen können, mussten nur die Teile des Code Selectors betrachtet werden, die solche Instruktionen erzeugen. Die erzeugten Load/Store Instruktionen der LLIR werden hier um Information darüber ergänzt, auf welche Datenobjekte sie zugreifen. Dies geschieht mit Hilfe von Pragmas, in denen der Name des referenzierten Objekts gespeichert wird. Natürlich werden nur für solche Instruktionen Pragmas generiert, die auch tatsächlich auf globale Variablen zugreifen.

Ein Problem ergibt sich an dieser Stelle bei der Verwendung von Zeigern. Die Variablen oder Speicherstellen, die ein Zeiger referenzieren kann, sind in der durch ICD-C gegebenen Repräsentation nicht bekannt, da diese Information auch in C-Programmen nicht explizit verfügbar ist. Die möglichen Ziele eines Zeigers ergeben sich aus dem Programmablauf. Um auch bei Verwendung von Zeigern korrekte Datenzugriffsinformationen für LLIR-Instruktionen bereitstellen zu können, ist also eine Berechnung der möglichen Ziele eines Zeigers notwenig. In ICD-C ist eine Aliasanalyse integriert, welche zu diesem Zweck entwickelt wurde [Bih05]. Der Code Selector wurde daher so erweitert, dass er diese Analyse aufruft und beim Übersetzen eines Zugriffes auf eine Zeigervariable die so berechneten Informationen nutzt, um das tatsächliche Ziel des Zugriffs zu bestimmen.

So besitzen alle vom Code-Selector erzeugten LLIR-Operationen, die auf globale Daten zu-

greifen, eine Referenz auf die Namen dieser Datenobjekte. Eine direkte Verknüpfung mit den *DataObject* Objekten der LLIR wird an dieser Stelle noch nicht erzeugt, da eine Instruktion auch auf Datenobjekte zugreifen kann, die in anderen LLIR-Instanzen definiert werden. Diese sind innerhalb des Code Selectors jedoch nicht bekannt. Auch die Information, ob ein lesender oder schreibender Zugriff erfolgt, wird hier noch nicht explizit bereitgestellt, dies kann jedoch aus dem Typ der Operation leicht bestimmt werden.

5.2.3 Anpassung der LLIR-CRL Konverter

Die Integration der statischen WCET Analyse in den WCC geschieht wie in Kapitel 3.3.3 beschrieben durch die Konvertierung der LLIR in das aiT-Eingabeformat CRL, den Aufruf der Analyse und schließlich der Rücktransformation der WCET-Informationen in die LLIR. Diese Aufgabe wird durch die WCC Komponente *LLIRAIT* erfüllt, in der für diese Arbeit einige Erweiterungen implementiert werden mussten.

Berechnung von Speicheradressen

Innerhalb der vom Code Selector erzeugten LLIR werden niemals absolute Adressen verwendet, um auf Variablen zuzugreifen. Dies liegt daran, dass die tatsächliche Adresse eines Objekts erst durch den Linker bestimmt wird. Eine Referenzierung von Datenobjekten erfolgt daher über so genannte Labels. Dies sind lediglich Zeichenketten, die den Namen des Objekts enthalten. Dieser muss nicht zwangsläufig mit dem Namen der Variablen im Quelltext des Programms übereinstimmen, da sonst bei statischen Variablen Redundanzen auftreten können. Ein Label bezeichnet also ein Datenobjekt innerhalb einer LLIR Instanz (und somit auch innerhalb des erzeugten Assemblercodes) eindeutig.

Normalerweise erhält aiT die Eingabe im CRL-Format aus einer Binärdatei, die bereits den Linker durchlaufen hat. Jedes Element besitzt dann eine feste, bekannte Adresse im Speicher, die Labels sind hier jedoch in der Regel nicht mehr vorhanden. Somit arbeitet aiT bei der statischen Analyse immer mit absoluten oder relativen Adressen. Für die Konvertierung der LLIR in das CRL-Format musste dieser Umstand berücksichtigt werden. Es soll nun zunächst die Situation vor Beginn dieser Arbeit beschrieben werden.

Um einen Aufruf von aiT zu ermöglichen, wurden innerhalb des Konverters beim Erzeugen von CRL-Elementen Pseudoadressen für diese berechnet. Diese mussten nicht tatsächlich mit den späteren Adressen im Programm übereinstimmen, es wurde lediglich ein Wert benötigt, den aiT während der Analyse nutzen kann. Zur Berechnung der Instruktionsadressen wurde so ein Zähler mit der Basisadresse des Standard-Speicherbereichs für Code initialisiert. Für jede konvertierte Instruktion wurde dieser Zähler entsprechend um deren Größe inkrementiert, so dass die Adressen zumindest den gleichen relativen Abstand zueinander besaßen wie im späteren Binary. Dies ermöglichte die Auflösung von relativen Sprungzielen durch aiT. Lediglich die Verwendung unterschiedlicher Speicher für Code wurde so nicht berücksichtigt. Diese war jedoch zu dem Zeitpunkt im WCC ohnehin noch nicht möglich.

Für Datenobjekte war eine solche Berechnung von Pseudoadressen nicht möglich, da vor Beginn dieser Arbeit im WCC keine Informationen über deren Größe zur Verfügung standen. Es konnte lediglich für ein Label bestimmt werden, ob dieses eine Stelle im Code oder ein Datum referenziert. Wann immer ein Datenlabel Parameter einer Operation war, wurde dieses durch

eine bestimmte feste Adresse im Speicher ersetzt. Für aiT lagen so alle Datenobjekte an der gleichen Speicherstelle. Abgesehen davon, dass so eine Unterscheidung der Zugriffszeiten bei der Verwendung unterschiedlicher Speicherbereiche nicht möglich war, konnte dieses Vorgehen auch Probleme bei der Wertanalyse bereiten. Bevor also die Implementierung eines Algorithmus zur Scratchpad-Allokation von Daten möglich war, musste zunächst die korrekte Behandlung von Datenobjekten durch aiT sichergestellt werden.

Das Ziel dieser Erweiterung ist es, die Anordnung von Daten im Speicher so zu rekonstruieren, wie sie im späteren Programm auftreten würde. Die Reihenfolge der Objekte innerhalb eines Speicherbereichs kann im Voraus allerdings nicht bestimmt werden, da diese durch den Linker geändert werden darf. Dies ist jedoch nicht von Bedeutung, da auch der C-Standard keine Annahmen über die Reihenfolge von Variablen im Speicher erlaubt. Wichtig ist nur, dass jedes Objekt in dem ihm zugewiesenen Speicherbereich liegt, eine feste Größe hat und, falls nötig, an einer bestimmten Speichergrenze ausgerichtet ist. Damit stehen aiT alle Informationen zur Verfügung, die für eine Abschätzung der WCET unter Berücksichtigung der Zugriffszeiten benötigt werden.

Vor der Konvertierung der LLIR in das CRL-Format wird daher eine Datenstruktur aufgebaut, die für jedes Datenobjekt dessen Adresse enthält. Gleichzeitig wird hier auch der Abruf des Objektes, das an einer bestimmten Speicherstelle liegt, ermöglicht. Implementiert wurde dies durch die C++ Klasse *MemoryManager*, welche den Aufbau des Speicherabbilds berechnet und einfachen Zugriff auf dieses bietet. Die Informationen können so in den Konvertern selbst ohne allzu große Anpassungen genutzt werden. Zunächst müssen alle LLIR-Instanzen, deren Datenobjekte berücksichtigt werden sollen, dem *MemoryManager* übergeben werden. Dann kann die Berechnung der Adressen durchgeführt werden.

Wie in Abschnitt 5.1.2 beschrieben, gehören Datenobjekte innerhalb der LLIR zu Sektionen. Für jeden Speicherbereich werden daher nacheinander alle Sektionen, die in diesen fallen, behandelt. Den einzelnen Datenobjekten darin wird dann jeweils die nächste freie Adresse zugeordnet. Dies geschieht unter Beachtung der für das Objekt definierten Ausrichtung. Integer-Variablen werden beispielsweise im WCC immer an 32-Bit Speichergrenzen ausgerichtet. Um ein korrektes Speicherabbild zu berechnen, muss also auch diese Information berücksichtigt werden.

Nach der Anwendung dieses Verfahrens für jeden Speicherbereich sind die Adressen aller Datenobjekte berechnet worden. Während der Konvertierung in das CRL Format können so die entsprechenden Labels durch die passende Adresse ersetzt werden, so dass aiT diese während der Analyse nutzen kann.

Inzwischen wurde im Zuge einer anderen Diplomarbeit auch für Instruktionsadressen ein solches Verfahren implementiert, so dass aiT nun komplett mit realistischen Adressen arbeiten kann. Dies ermöglicht die WCET-Abschätzung für Programme, deren Daten und Code in unterschiedlichen Speicherbereichen liegen und bildet die Grundlage für alle Optimierungen, die sich mit dem Auslagern von LLIR-Elementen in die Scratchpad-Speicher des TriCore beschäftigen.

Durch die Kenntnis aller Adressen, die Daten enthalten, kann aiT eine zusätzliche Hilfestellung bei der statischen Analyse gegeben werden. Innerhalb der CRL kann eingeschränkt werden, auf welche Speicheradressen Datenzugriffe des Programms erfolgen. So kann der Rechenaufwand innerhalb der Wertanalyse verringert werden, da aiT nun nicht mehr den gesamten Speicher als Ziel in Betracht ziehen muss, sondern nur noch die Bereiche, in denen Stack, CSA und globale Daten liegen.

Berechnung von Datenzugriffen durch aiT

Während der Wertanalyse verfolgt aiT den Inhalt der Register, um so bei einem Speicherzugriff mögliche Zieladressen berechnen zu können. Für jedes Register wird an jeder Stelle des Programms ein Wertebereich bestimmt, in dem dessen Inhalte liegen. Bei allen Load/Store Operationen, die ein Register zur Adressierung des Speichers nutzen, werden diese Informationen genutzt, um den Speicherbereich zu bestimmen, auf den zugegriffen wird. Dieser wird in späteren Stufen benötigt, um die Ausführungszeit der entsprechenden Instruktionen zu berechnen.

Die Berechnung der möglichen Registerwerte ist äußerst komplex, da alle Operationen, die einen Effekt auf das Register haben, berücksichtigt werden müssen. Diese können jedoch wiederum von Speicherinhalten oder anderen Registern abhängen. Teilweise kann aiT daher den Inhalt der Register nicht bestimmen. Als Wertebereich wird dann der gesamte mögliche Inhalt des Registers betrachtet. Dies hat zur Folge, dass für Load/Store Operationen, die dieses Register zur Adressierung des Speichers nutzen, die Zieladresse nicht berechnet werden kann. Da aiT immer sichere WCET-Abschätzungen liefern muss, muss in diesem Fall von einem Zugriff auf den langsamsten Speicher ausgegangen werden. Dies ist ungünstig, da in einem solchen Fall die Auslagerung eines Objektes in den schnelleren Scratchpad-Speicher gar nicht berücksichtigt würde.

Durch die im Code Selector gewonnenen Datenzugriffsinformationen, kann aiT hier jedoch unterstützt werden. Die CRL ermöglicht es, für einzelne Instruktionen den Speicherbereich, auf den diese zugreifen, zu bestimmen. Durch entsprechende Annotationen in der CRL-Datei können also die im WCC bekannten Zugriffsziele auch an aiT weitergereicht werden. Auch hier ist wieder die Angabe absoluter Speicheradressen erforderlich. Diese sind durch den *MemoryManager* bereits bekannt und können so direkt genutzt werden. Wird eine so modifizierte CRL-Datei durch aiT analysiert, wird für die einzelnen Load/Store Operationen nur der angegebene Speicherbereich als Ziel in Betracht gezogen. Deren Ausführungszeiten werden also korrekt berechnet.

Durch die Wertanalyse werden alle erkannten Speicherzugriffe in der CRL-Datei notiert. Hierzu existieren zwei CRL-Attribute für Instruktionen, die einen lesenden bzw. schreibenden Zugriff kennzeichnen. Wurden im Vorfeld wie oben beschrieben die möglichen Zugriffsorte eingeschränkt, so enthalten diese Attribute Teilmengen der angegebenen Speicherbereiche. Es kann vorkommen, dass aiT genauere Informationen ermitteln kann, als dies im Code Selector möglich war, zum Beispiel, wenn nur auf einige Elemente eines Arrays zugegriffen wird. Die Werte innerhalb der CRL sind somit immer mindestens so genau wie die im Code Selector ermittelten. Daher werden die dort erzeugten Pragmas nun nicht mehr benötigt.

Da dem *MemoryManager* sämtliche Datenobjekte des zu übersetzenden Programms bekannt sind, kann nun auch für Zugriffe auf Daten, die zu einer anderen LLIR-Instanz gehören als die zugreifende Instruktion, das entsprechende *DataObject* Objekt ermittelt werden. Dadurch ist nun eine direkte Verknüpfung zwischen Instruktionen und den davon referenzieren Datenobjekten möglich.

Bereitstellen der Datenzugriffsinformationen

Nach einem erfolgreichen aiT-Durchlauf werden die Elemente der LLIR um die berechneten WCET-Daten ergänzt. Dies geschieht durch den *Objective Handler* Mechanismus der LLIR. Auf die gleiche Weise sollen nun auch die Datenzugriffsinformationen an die entsprechenden LLIR-

Elemente gehängt werden. Dazu wurde die Klasse *DataAccess* implementiert, die für ein LLIR-Element den einfachen Abruf von Datenzugriffsinformationen erlaubt. Dies soll nicht nur für Instruktionen, sondern auch für Basisblöcke und Funktionen ermöglicht werden. Daher werden auch mehrmalige Zugriffe auf ein Datenobjekt unterstützt. Für einen Basisblock würde so die Summe aller Zugriffe der in ihm enthaltenen Instruktionen zurückgegeben. Konkret können folgende Informationen zur Verfügung gestellt werden:

- Anzahl der lesenden Zugriffe pro Datenobjekt
- Anzahl der schreibenden Zugriffe pro Datenobjekt
- Anzahl der Zugriffe (lesend oder schreibend) pro Datenobjekt

Jeder LLIR-Instruktion, die auf den Speicher zugreift, wird eine Instanz des *DataAccess Objectives* angehängt. In dieser werden die jeweiligen Speicherzugriffe gespeichert. Dazu müssen zu den von aiT ermittelten Speicherbereichen die an diesen Stellen liegenden Datenobjekte ermittelt werden. Auch diese Aufgabe wird durch den *MemoryManager* erledigt. Daraufhin werden diese Daten für jeden Basisblock zusammengefasst und an diesen gehängt. Schließlich werden auch für jede Funktion die Zugriffe aller in ihr enthaltenen Blöcke bereitgestellt.

Nach der WCET-Analyse stehen somit zu den LLIR-Elementen detaillierte Informationen über die darin enthaltenen Datenzugriffe zur Verfügung, die von allen folgenden Stufen des Compilers genutzt werden können. Im Moment ist die Scratchpad-Allokation von Daten jedoch die einzige Komponente, welche Gebrauch von diesen Daten macht.

5.3 Modellierung des Kontrollflussgraphen

Zur Formulierung des ILP werden eine Reihe von Informationen über die Struktur des Programms benötigt, die sich aus der durch die LLIR gegebenen Beschreibung nicht ablesen lassen. Sie lassen sich jedoch daraus berechnen, so dass keine weiteren Eingabedaten außer den mit WCET- und Datenzugriffsinformationen angereicherten LLIR Funktionen und Basisblöcken nötig sind, um den Kontrollflussgraphen in einer Form aufzubauen, die eine einfache Erzeugung der Constraints erlaubt. Dazu muss auch die Verschachtelung von Schleifen, die eine wichtige Rolle für den Algorithmus spielt, abgebildet werden. Erreicht werden soll also eine Darstellung des Kontrollflusses, entlang derer die ILP-Formulierung ohne weitere Transformationen aufgebaut werden kann.

5.3.1 Datenstruktur zur Darstellung des Kontrollflussgraphen

Es wurde eine Datenstruktur entwickelt, die den Kontrollfluss des Programms als Graph abbildet und dabei folgende Funktionalität bietet:

- Kennzeichnung des Programmeintrittspunktes
- Repräsentation von Funktionen und der in ihnen liegenden Basisblöcke
- Bereitstellen von Vorgänger- und Nachfolgerbeziehungen zwischen den Knoten

- Darstellung eines Call-Graphen des Programms zur Erkennung von Rekursion
- Möglichkeit der Kontraktion aller Knoten innerhalb einer Schleife zu einem neuen Knoten

Da die Implementierung der Optimierung in der Programmiersprache C++ erfolgte, konnte das Prinzip der Vererbung ausgenutzt werden, um gleichartige Funktionen in Oberklassen zu sammeln und so eine logische Strukturierung zu erhalten und Redundanzen zu vermeiden. Abbildung 5.5 zeigt das Klassendiagramm des Kontrollflussgraphen. Einige Beziehungen zwischen den Klassen, die für eine Beschreibung an dieser Stelle nicht relevant sind, wurden in dem Diagramm bewusst ausgelassen.

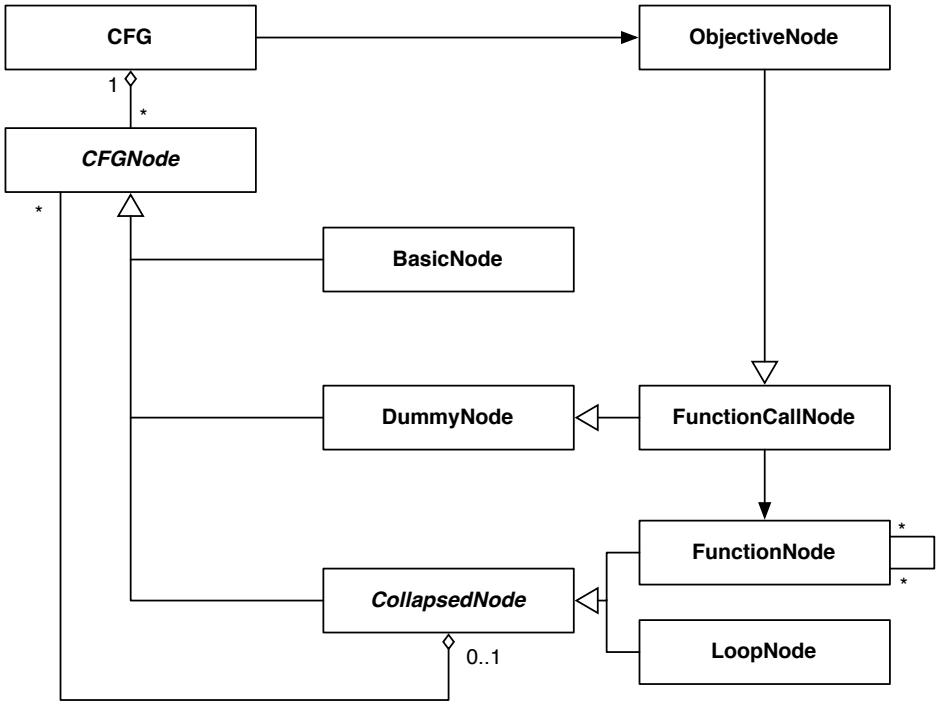

Abbildung 5.5: Klassendiagramm der Datenstrukturen zur Darstellung des Kontrollflussgraphen

Die zentrale Klasse stellt *CFG* dar. Diese repräsentiert den Kontrollflussgraphen und bietet Zugriff auf alle Knoten, die zu diesem gehören. Alle weiteren dargestellten Klassen stellen Knoten unterschiedlicher Typen dar. Die Erzeugung von Instanzen der Knoten kann nur über die *CFG*-Klasse erfolgen. Diese übernimmt so auch die Freigabe des belegten Speichers, sobald der Graph nicht mehr benötigt wird.

Jeder Knoten im Graphen erbt von der abstrakten Basisklasse *CFGNode*. Diese bietet Zugriff auf die Daten, die für alle Knoten benötigt werden. Dazu gehören ein Name zur Identifizierung, der Typ des Knotens und die jeweiligen Vorgänger und Nachfolger. Durch letztere werden die Kanten des Graphen beschrieben.

Die Klasse *BasicNode* repräsentiert einen Basisblock des Programms. Sie ist das direkte Gegenstück zu dem zugehörigen Block in der LLIR und speichert als solches auch eine direkte Referenz auf diesen. So kann über die *BasicNode* auf alle innerhalb der LLIR verfügbaren Informationen inklusive der Datenzugriffe und WCET-Daten zugegriffen werden.

Durch die abstrakte Klasse *CollapsedNode* wird ein Knoten beschrieben, der selbst wieder einen Untergraph enthalten kann. Dazu wird Zugriff auf die Menge aller Unterknoten geboten. Auch der Startknoten des enthaltenen Graphen kann hier abgefragt werden. Zudem lässt sich hier die Tiefe der Verschachtelung abfragen.

Eine konkrete Unterklasse von *CollapsedNode* stellt *LoopNode* dar. Diese kann genutzt werden, um eine Schleife im Kontrollflussgraphen zu repräsentieren. So können die Knoten einer Schleife zusammengefasst werden, ohne die Information über den Kontrollfluss innerhalb der Schleife zu verlieren. Für den umliegenden Graphen besteht eine Schleife dann nur noch aus einem Knoten, bei Bedarf können jedoch auch die Knoten innerhalb der Schleife betrachtet werden. Innerhalb des Untergraphen einer Schleife können natürlich ebenfalls Schleifenknoten enthalten sein. Dies ermöglicht die Darstellung beliebig tief verschachtelter Schleifen.

Auch die Klasse *FunctionNode*, welche für eine Funktion des Programms steht, erbt von *CollapsedNode*. Neben dem Untergraphen mit allen Knoten, die zu der Funktion gehören, enthält sie eine Referenz auf das zugehörige LLIR-Element. Zudem kann hier eine Liste aller innerhalb des Knotens aufgerufenen Funktionen abgerufen werden. Damit stellt die Datenstruktur zusätzlich zum Kontrollflussgraphen des Programms auch einen Call-Graph dar, mit dessen Hilfe sich Rekursion erkennen lässt.

Die Klasse *DummyNode* erlaubt die Erzeugung eines Knotens ohne weiteren Effekt und insbesondere ohne ein entsprechendes Gegenstück innerhalb der LLIR. Diese kann beispielsweise genutzt werden, um eindeutige Start- oder Endknoten eines Untergraphen zu erzeugen, falls davon sonst mehrere existieren würden.

Zur Darstellung von Funktionsaufrufen existiert die Klasse *FunctionCallNode*. Diese besitzt ebenfalls kein Gegenstück innerhalb der LLIR, da Funktionsaufrufe hier nur als Instruktionen innerhalb von Basisblöcken auftreten. Während der Implementierung des Algorithmus zeigte sich, dass die Darstellung als eigener Knoten des Graphen einen deutlich vereinfachten Umgang mit Funktionsaufrufen ermöglichte, daher wurde diese Klasse (als Unterklasse der *DummyNode*) eingeführt. Sie speichert einen Zeiger auf das *FunctionNode* Objekt der aufgerufenen Funktion.

Da eine *FunctionCallNode* keine Referenz zu einem LLIR-Basisblock besitzt, und somit auch keine Instruktionen enthalten kann, befindet sich der tatsächliche Funktionsaufruf immer innerhalb des vorhergehenden Knoten. Dieser muss dementsprechend der einzige Vorgänger und vom Typ *BasicNode* sein. Die Instruktion, welche tatsächlich den Funktionsaufruf durchführt, befindet sich dann ganz am Ende dieses Basisblocks.

Innerhalb eines Graphen existiert genau eine *ObjectiveNode*. Diese kennzeichnet den Eintrittspunkt des Programms. Da dies als Aufruf der Startfunktion behandelt werden kann, erbt sie von der Klasse *FunctionCallNode*. Der Name *ObjectiveNode* ergibt sich daraus, dass anhand dieser Klasse bei der Formulierung des ILP die Zielfunktion gebildet wird.

Zusätzlich stellt der Graph selbst noch eine Reihe von Hilfsfunktionen bereit, die den Aufbau einer solchen Struktur unterstützen. Insbesondere das Zusammenfassen der Knoten einer Schleife kann so durch nur einen Aufruf erledigt werden. Dafür müssen nur der Schleifenkopf und die Knoten des Schleifenkörpers an eine Funktion übergeben werden, welche dann den Umbau der Struktur innerhalb des Graphen vornimmt.

Kontrollflussgraph eines Beispielprogramms

Die Struktur eines so dargestellten Kontrollflussgraphen soll nun an einem Beispiel erläutert werden. In Abbildung 5.6 ist ein Programm mit zwei Funktionen, *main* und $func1$ zu sehen. Instanzen der Klasse *BasicNode* sind als Kreise dargestellt, für alle anderen Elemente ist der entsprechende Klassenname kursiv gekennzeichnet.

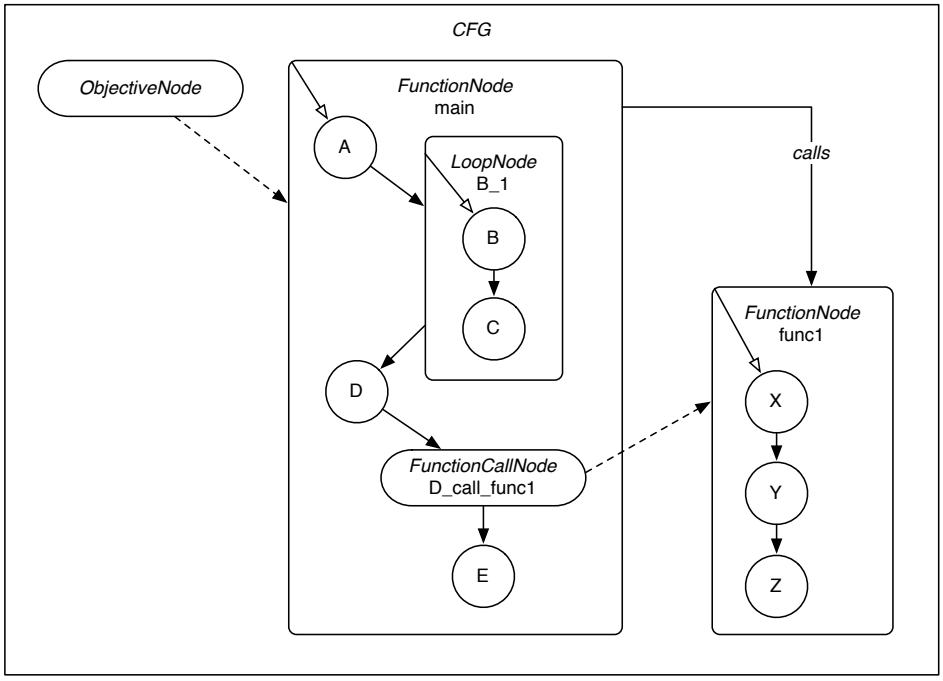

Abbildung 5.6: Beispiel für die Struktur eines *CFG*

Die Startfunktion main wird durch die *ObjectiveNode* des *CFG* gekennzeichnet. Darin enthalten ist sowohl eine Schleife als auch ein Funktionsaufruf. Letzterer ist zum einen als direkte Beziehung zwischen den Funktionsknoten ersichtlich, zum anderen durch die *FunctionCallNode*. Diese enthält eine Referenz auf $func1$, obwohl der eigentliche Aufruf der Funktion bereits innerhalb des Basisblocks D geschieht. Da Funktionsaufrufe jedoch nur als letze Instruktion eines Basisblocks auftreten können, ist der durch den Graphen dargestellte Ablauf korrekt.

Die Schleife wird durch eine *LoopNode* dargestellt. Diese tritt innerhalb von *main* als einzelner Knoten auf. In diesem Knoten enthalten sind die beiden Basisblöcke, die zu der Schleife gehören. Die Rückkante der Schleife ($C \rightarrow B$) wurde während der Schleifenkontraktion entfernt. Somit existieren in den Untergraphen der Funktionen keine Zyklen mehr, was für die Formulierung des ILP von Bedeutung ist.

Mit dieser Darstellung steht eine vielseitige und erweiterbare Datenstruktur bereit, entlang derer die Formulierung des ILP sehr einfach vorgenommen werden kann. Zunächst muss dafür jedoch die LLIR in diese Darstellung übersetzt werden. Dies geschieht direkt im Konstruktor der Klasse *CFG*, dem eine Liste von LLIR-Instanzen und der Name der Startfunktion übergeben wird. Der Aufbau des Graphen wird im folgenden Abschnitt behandelt.

5.3.2 Erzeugen des initialen Kontrollflussgraphen

Die Erzeugung des Kontrollflussgraphen geschieht in mehreren Schritten. Zunächst wird für jede Funktion ein leerer Knoten des Typs *FunctionNode* erstellt. Somit ist bereits für jede Funktion ein Knoten vorhanden, der bei der Verknüpfung von Funktionsaufrufen später genutzt werden kann. Der so erzeugte Graph hat bereits jetzt den Vorteil, eine globale Sicht auf das gesamte Programm zu bieten, während die einzelnen LLIR-Instanzen nur die Funktionen jeweils einer Quelltextdatei enthalten. Daraufhin werden für jeden Funktionsknoten die folgenden Schritte durchgeführt:

1. Erzeugung von *BasicNodes*
   Für jeden Basisblock der Funktion wird ein Knoten erstellt und die Referenz auf den zugehörigen LLIR Block gespeichert. Die so erzeugten Knoten werden alle dem Untergraphen des Funktionsknotens hinzugefügt, es werden allerdings noch keine Kanten zwischen den Knoten erzeugt.

2. Hinzufügen der Kanten zwischen *BasicNodes*
   Erst im zweiten Schritt können die Kanten zwischen den Knoten hinzugefügt werden, da erst jetzt für jeden Basisblock ein Knoten existiert. Die jeweiligen Vorgänger und Nachfolger eines Blocks werden dabei der LLIR entnommen. Zu diesem Zweck bietet die Klasse *CFG* eine Hilfsfunktion, die für einen LLIR-Basisblock den zugehörigen CFG-Knoten ermittelt.

3. Einfügen von *FunctionCallNodes*
   Die Knoten zur Repräsentation von Funktionaufrufen haben keine äquivalente Darstellung in der LLIR. Daher müssen die Instruktionen jedes Blocks untersucht werden. Wird innerhalb eines Knotens ein Funktionsaufruf identifiziert, so erhält dieser als einzigen Nachfolger eine neu erzeugte *FunctionCallNode* mit der Referenz auf die aufgerufene Funktion.

   Da der Funktionsaufruf selbst die Kontrollflussverzweigung am Ende eines Blocks darstellt, besitzen Basisblöcke, die einen Funktionsaufruf enthalten, maximal einen Nachfolger. Dieser Knoten wird dann zum Nachfolger der *FunctionCallNode*. Dies ist in Abbildung 5.7 für einen Aufruf der Funktion *foo* im Block *A* dargestellt.

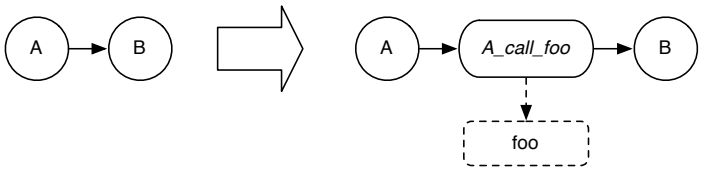

Abbildung 5.7: Einfügen einer *FunctionCallNode*

Zu diesem Zeitpunkt besitzt der Graph außer der expliziten Verdeutlichung von Funktionsaufrufen noch keine Unterschiede gegenüber dem Inhalt der LLIR. Nun kann der Call-Graph aufgebaut werden. Dazu wird eine Tiefensuche auf den Funktionsknoten, startend bei der Startfunktion, durchgeführt. Für jede *FunctionCallNode* wird die durch diese aufgerufene Funktion besucht und als Nachfolger der aufrufenden Funktion hinzugefügt. Wird hierbei ein Zyklus erkannt, liegt ein rekursiver Aufruf vor. Dies wird in der *FunctionCallNode* vermerkt, damit der Aufruf bei der Formulierung des ILP entsprechend behandelt werden kann.

Der Graph besitzt nun alle Informationen, liegt aber immer noch in einer flachen Struktur vor. Das heißt, innerhalb der Funktionen existieren bisher lediglich Knoten für Basisblöcke und Funktionsaufrufe. Es fehlt die Kontraktion der Schleifen, um eine verschachtelte Struktur mit zyklenfreien Untergraphen zu erhalten.

5.3.3 Iterative Verschachtelung

Es wird nun nacheinander für jede Funktion des Programms der zugehörige Untergraph behandelt. Da eine beliebig tiefe Verschachtelung von Schleifen möglich ist, muss jeweils auf der untersten Ebene begonnen werden, diese zu behandeln. Die Erzeugung von Schleifenknoten wird daher startend bei den innersten Schleifen iterativ durchgeführt, bis der Graph der Funktion zyklenfrei ist, also alle Schleifen beseitigt wurden. Die Idee hinter diesem Vorgehen wurde schon während der Beschreibung der ILP-Formulierung im Kapitel 4.4.1 erklärt. Aus diesem Grund wird an dieser Stelle nur kurz auf die Implementierung eingegangen.

Startend bei dem nicht verschachtelten Graphen der Funktion werden abwechselnd die folgenden zwei Schritte durchgeführt.

1. Schleifensuche im Graphen
   Es wurde ein Algorithmus implementiert, der alle Schleifen innerhalb einer Ebene des Graphen finden und die Knoten des Schleifenkörpers dem entsprechenden Schleifenkopf zuordnen kann. Zusätzlich berechnet dieser die Verschachtelungstiefe der Schleifen. Mit Hilfe dieser Information können die innersten Schleifen identifiziert werden. Wird keine Schleife mehr gefunden, kann die Behandlung abgebrochen werden.

   Die Funktionsweise des Algorithmus zur Schleifensuche wird weiter unten erläutert.

2. Kontraktion der innersten Schleifen
   Für die im ersten Schritt gefundenen innersten Schleifen wird nun je eine *LoopNode* erstellt, der die zugehörigen Knoten hinzugefügt werden. Nun müssen die Kanten angepasst werden. Zunächst werden alle Rückkanten der Schleife entfernt. Somit sind die Untergraphen der Schleifenknoten zyklenfrei, da innerhalb der innersten Schleifen keine weiteren Kanten einen Zyklus erzeugen können.

   Als nächstes werden alle Kanten, die auf einen Knoten innerhalb der Schleife zeigen, auf den neu erstellten Knoten umgelenkt. Dies ermöglicht auch die Behandlung von Schleifen mit mehreren Eintrittspunkten, hier geht allerdings die Information verloren, an welcher Stelle der Eintritt erfolgt. Analog wird für alle Kanten aus der Schleife heraus verfahren. Diese gehen nun nicht mehr von den Knoten innerhalb der Schleifen, sondern von der *LoopNode* aus. Im Fall von multiplen Austrittspunkten kann auch hier nicht mehr bestimmt werden, an welcher Stelle der Austritt erfolgt. Um diese Information nicht zu verlieren, wird innerhalb des Knotens vermerkt, dass es sich um einen Austrittspunkt handelt.

   Abbildung 5.8 zeigt die Kontraktion einer Schleife mit einem Eintrittspunkt und zwei Austrittspunkten. Nach der Transformation kann anhand der Kanten nicht mehr erkannt werden, dass auch über den Knoten D ein Verlassen der Schleife möglich ist, im Knoten D selbst ist aber gespeichert, dass es sich um einen Austrittsknoten handelt.

Wurden in Schritt 1 keine weiteren Zyklen mehr gefunden, so sind für alle Schleifen der Funktion

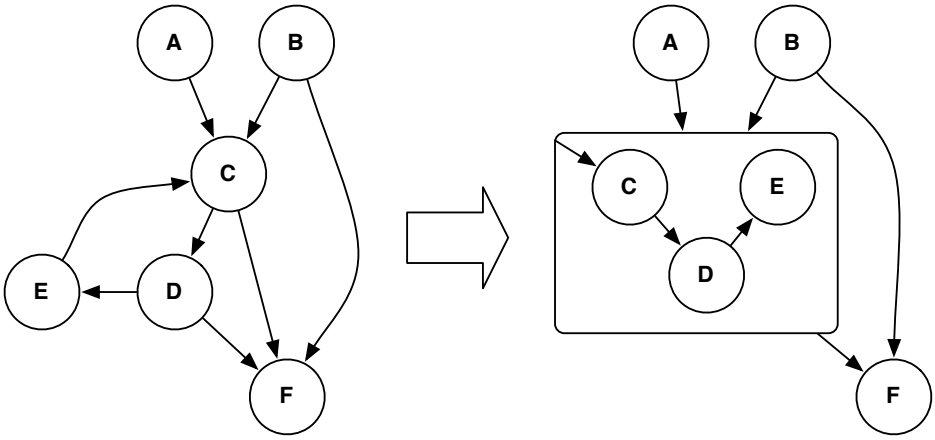

Abbildung 5.8: Beispiel einer Schleifenkontraktion

entsprechende Knoten erstellt worden. Der Graph hat nun eine Form, entlang der die Constraints der ILP-Formulierung gebildet werden können.

Schleifenerkennung

Es bleibt nun noch zu klären, wie innerhalb eines Kontrollflussgraphen effizient alle Schleifen erkannt werden können. Das Problem ist deutlich komplexer als die simple Suche von Zyklen in einem Graphen, da die Knoten eines Schleifenkörpers genau der zugehörigen Schleife, also dem innersten Schleifenkopf zugeordnet werden müssen. Eine Methode, diese Informationen einem dekompilierten Programm mit nahezu linearem Zeitaufwand zu entnehmen, stellen Wei, Mao, Zou und Chen in [WMZC07] vor. Diese wurde für die vorliegende Arbeit implementiert.

Der Algorithmus benötigt zur Erkennung von Schleifen nur einen Durchlauf einer Tiefensuche auf dem Kontrollflussgraphen. Für jeden Knoten des Graphen wird der Kopf der innersten Schleife gespeichert, zu deren Körper er gehört. Anhand dieser Informationen lassen sich im Anschluss alle Schleifen des Programms identifizieren. Auch Schleifen mit mehreren Eintritts- und Austrittspunkten werden gefunden und korrekt behandelt.

Die Tiefensuche startet im ersten Knoten einer Funktion und folgt an jedem Knoten nacheinander den ausgehenden Kanten. In Abbildung 5.9 sind die verschiedenen Fälle illustriert, die bei der Behandlung einer Kante des Kontrollflussgraphen eintreten können. Der aktuelle Knoten ist hierbei als C gekennzeichnet, der jeweils betrachtete Nachfolger als N. Bereits besuchte Knoten sind grau markiert, und der aktuelle Pfad des DFS-Durchlaufs ist durch fett gedruckte Pfeile hervorgehoben.

- **Fall 1** Wurde der Nachfolger N zuvor nicht besucht, so wird für diesen ein rekursiver Aufruf gestartet. Bei der Rückkehr wird geprüft, ob N einem Schleifenkörper zugeordnet wurde. Falls dies zutrifft, wird auch C dieser Schleife zugeordnet, es sei denn, C selbst ist der Schleifenkopf.
- **Fall 2** Wurde N bereits besucht und liegt im aktuellen Pfad, so wird N als Schleifenkopf von C markiert. Damit ist C der entsprechenden Schleife zugeordnet.

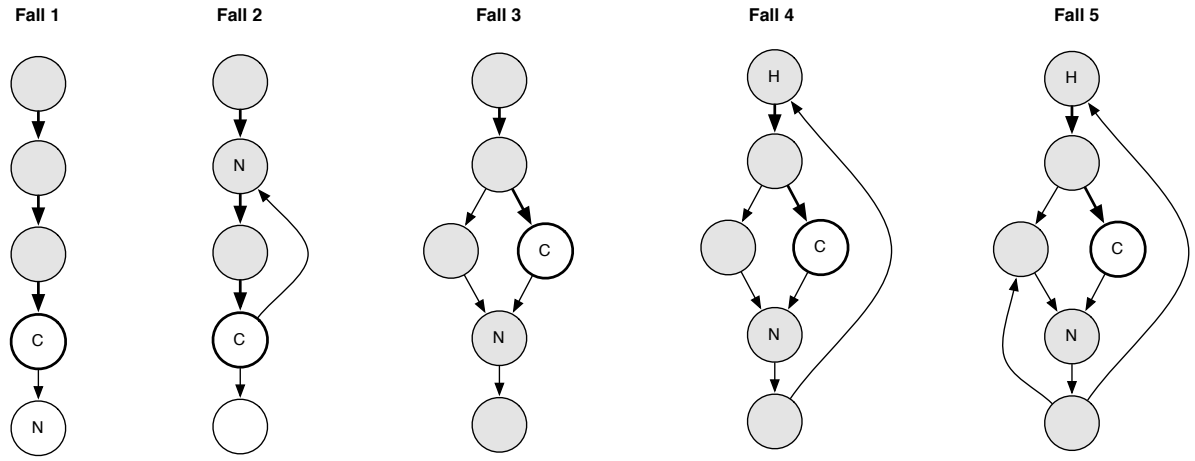

Abbildung 5.9: Unterschiedliche Fälle bei der Schleifenerkennung

- **Fall 3** Wurde N bereits besucht, liegt aber weder im aktuellen Pfad noch innerhalb einer bereits gefundenen Schleife, so wird lediglich eine Kontrollflussverzweigung wieder zusammengeführt. In diesem Fall ist keine weitere Behandlung erforderlich.
- **Fall 4** Auch in diesem Fall wurde N bereits besucht und liegt nicht im aktuellen Pfad. Allerdings gehört der Knoten bereits zu einer Schleife, deren Schleifenkopf im aktuellen Pfad liegt. Damit gehört auch C zu dieser Schleife.
- **Fall 5** Wurde N bereits besucht und gehört zu einer Schleife, deren Kopf nicht im aktuellen Pfad liegt, dann ist die Kante $C \rightarrow N$ ein weiterer Einstiegspunkt in die Schleife, in der N liegt. In diesem Fall wird ausgehend von N nach dem innersten Schleifenkopf gesucht, der innerhalb des aktuellen Pfades liegt. Im Beispiel ist dies der Knoten H, der C somit als Schleifenkopf zugeordnet wird. Wird kein solcher Knoten gefunden, gehört C selbst zu keiner Schleife.

Mit Hilfe dieses Vorgehens können beliebig tief verschachtelte Schleifen im Kontrollflussgraphen gefunden werden. Daraus lässt sich auch die Information über die Verschachtelungstiefe des Graphen ableiten, so dass die innersten Schleifen identifiziert und im Rahmen der iterativen Verschachtelung behandelt werden können.

5.4 Formulierung und Lösung des ILP

Innerhalb des Graphen liegt nun das Programm in einer Form vor, die direkt zur Erzeugung der ILP-Constraints geeignet ist. Zur Formulierung des Problems wird die in den WCC integrierte *libilp* genutzt, eine am Lehrstuhl entwickelte Bibliothek, die den Aufbau eines ILP über eine objektorientierte Schnittstelle erlaubt. Die *libilp* stellt C++ Klassen zur Definition von Constraints und Variablen bereit und ermöglicht die Lösung durch einen externen LP-Solver.

Die zu erzeugenden Constraints sind in Kapitel 4 bereits beschrieben worden. Diese müssen nun für das entsprechende Programm formuliert werden. Für jedes Datenobjekt wird so zu-

nächst die Entscheidungsvariable (S_d) definiert. Dann kann die Beschränkung der Scratchpad-Kapazität durch eine Restriktion ausgedrückt werden. Dessen Größe (*scratchpad_size*) ist in der Speicherlayout-Konfigurationsdatei definiert und steht somit im WCC zur Verfügung.

Alle weiteren Constraints und Variablen können anhand des Graphen erzeugt werden. Durch dessen verschachtelte Struktur ist hier ein rekursives Vorgehen erforderlich. Dabei müssen für jeden Knoten dessen Kosten (C_n) und Gewicht (W_n) definiert werden. Für diese Aufgabe ist die Funktion *createConstraints* verantwortlich. Dieser wird ein Knoten des Graphen übergeben. Je nach dessen Typ wird dann zunächst unterschiedlich verfahren:

- *FunctionNode*
  In jedem Funktionsknoten befindet sich ein Untergraph, der den Kontrollfluss der Funktion wie im letzten Abschnitt beschrieben repräsentiert. Daher wird an dieser Stelle für jeden Knoten dieses Graphen ein rekursiver Aufruf gestartet. Danach können die Kosten der Funktion als das Gewicht des Startknotens des Untergraphen definiert werden.

- *BasicNode*
  Da eine *BasicNode* direkt für einen Basisblock steht, müssen hier die Kosten des Knotens bestimmt werden. Diese sind definiert als die WCET des Blocks abzüglich des Gewinns, der in diesem Block durch die Scratchpad-Allokation erzielt werden kann. Wie die Werte der hierfür benötigten Konstanten gewonnen werden, wird im nächsten Abschnitt dieses Kapitels erläutert.

- *FunctionCallNode*
  Im Normalfall entsprechen die Kosten eines Knotens vom Typ *FunctionCallNode* denen der aufgerufenen Funktion. Falls diese noch nicht behandelt wurde, wird an dieser Stelle die entsprechende Variable bereits erzeugt und gesichert, so dass sie später abgerufen werden kann.

  Einen Sonderfall stellen rekursive Funktionen dar. Hier muss für den ersten Aufruf die maximale Rekursionstiefe mit den Funktionskosten multipliziert werden. Alle rekursiven Aufrufe können dann ignoriert werden, für diese werden die Kosten des Funktionsaufrufs also auf 0 gesetzt.

- *LoopNode*
  Für Schleifen ist zunächst ein rekursiver Aufruf von *createConstraints* für alle Knoten innerhalb der Schleife (analog zur Behandlung eines Funktionsknotens) nötig. Im Anschluss werden die Kosten des Knotens definiert als das Gewicht des ersten Schleifenknotens multipliziert mit der maximalen Iterationshäufigkeit der Schleife.

- *DummyNode*
  Dieser Knotentyp hat keine Auswirkungen auf den Programmverlauf. Daher werden die Kosten auf 0 gesetzt.

Nun, da die Kosten des aktuellen Knotens definiert sind, werden für alle ausgehenden Kanten (bzw. den Knoten selbst, falls er keine Nachfolger besitzt) die zugehörigen Kontrollflussconstraints erzeugt (siehe Kapitel 4.4) und so das Gewicht des Knotens definiert. Dies erfolgt für jeden Knotentyp auf die gleiche Art und Weise.

Dank des rekursiven Verhaltens reicht es, einmal für jede *FunctionNode* die Funktion *createConstraints* aufzurufen. Danach sind alle Constraints erzeugt worden und es fehlt nur noch die

Definition der Zielfunktion. Diese fordert die Minimierung der Kosten der Startfunktion des Programms, welche durch die *ObjectiveNode* gekennzeichnet ist. Wurde diese erzeugt, ist das ILP vollständig formuliert und kann gelöst werden.

5.4.1 Konstanten in der ILP-Formulierung

Innerhalb der ILP-Formulierung treten drei Arten von Konstanten auf, deren Herkunft bis jetzt noch nicht geklärt wurde:

- Die WCET der einzelnen Basisblöcke ($wcet_b$)
- Die maximale Iterationshäufigkeit von Schleifen ($wcec_b$)
- Der Gewinn, welcher durch die Auslagerung eines Datenobjekts in den Scratchpad-Speicher für einen Basisblock erzielt wird ($gain_{b,d}$)

Nach dem Aufruf der WCET-Analyse befinden sich alle von aiT berechneten Daten innerhalb der LLIR und stehen somit während der Generierung des ILP zur Verfügung. Da aiT jedoch WCET-Informationen an Kanten, nicht an Basisblöcken, speichert, existiert keine direkte WCET-Abschätzung pro Block. Hinzu kommt, dass aiT das Konzept unterschiedlicher Ausführungskontexte unterstützt und somit für jede ausgehende Kante eines Blocks in jedem Kontext ein Wert zur Verfügung steht. Da das ILP keine Berücksichtigung unterschiedlicher Kontexte erlaubt, muss aus den durch aiT berechneten WCET-Werten der Kanten eine Block-WCET gewonnen werden. Dazu wird in allen Kontexten die Kante mit der höchsten WCET gesucht und deren WCET für den Block übernommen. Diese stellt eine eventuell etwas ungenauere, aber in jedem Fall sichere Abschätzung der maximalen Ausführungszeit des Blocks dar.

Die maximale Anzahl von Schleifeniterationen wird durch den *Flow Fact* Mechanismus des WCC [Sch07] bereitgestellt und stammt aus Benutzerannotationen im Quelltext oder einer automatischen Schleifenanalyse [Cor08]. Nach der WCET-Analyse werden diese Daten um eventuell von aiT gefundene Schleifengrenzen ergänzt und innerhalb der LLIR Basisblöcke gespeichert. An dieser Stelle muss eine Besonderheit des *Flow Fact* Mechanismus beachtet werden: Der für eine Schleife gespeicherte Wert entspricht aus technischen Gründen der Anzahl der Überprüfungen der Abbruchbedingung. Für `while` Schleifen beispielsweise, deren Abbruchbedingung vor jedem Durchlauf getestet wird, ist dieser Wert also um 1 höher als die tatsächliche Anzahl der Schleifeniterationen. Bei der Definition der Kosten einer Schleife wird daher überprüft, ob im Schleifenkopf ein Austritt aus der Schleife möglich ist (dafür wurden bei der Schleifenkontraktion die Austrittsknoten entsprechend markiert). Falls dies zutrifft, so wird die Austrittsbedingung auch vor dem Betreten der Schleife geprüft und der durch den *Flow Fact* definierte Wert muss um 1 verringert werden.

Die Berechnung des Gewinns, der für einen Basisblock bei Auslagerung eines bestimmten Datenobjekts erzieht wird, ist nicht so leicht wie die beiden zuvor beschriebenen Konstanten zu ermitteln. Dieses Problem soll im folgenden Abschnitt erläutert werden.

5.4.2 Berechnung des Blockgewinns bei Auslagerung eines Datenobjekts

Um die Auslagerung von Datenobjekten in den Gewichten der Knoten zu berücksichtigen, ist die Kenntnis des Gewinns pro Block und ausgelagertem Objekt nötig. In Kapitel 4 wird dieser Wert als $gain_{b,d}$ bezeichnet. Im Idealfall könnte er ermittelt werden, indem die Anzahl der Zugriffe auf das Objekt innerhalb des Blocks mit der Geschwindigkeitsdifferenz zwischen Scratchpad und SRAM multipliziert wird. Durch die komplexe Pipeline des TriCore Prozessors wird eine solch einfache Berechnung jedoch zumindest für schreibende Zugriffe verhindert. Während das Anfordern eines Datums aus dem Speicher in jedem Fall die Pipeline anhält, bis der gelesene Wert vorliegt, können einzelne Schreibzugriffe im Hintergrund durchgeführt werden, falls der Datenbus (DLMB) frei ist. In einem solchen Fall ist für die entsprechende Instruktion auch bei Auslagerung des referenzierten Datenobjekts in den Scratchpad-Speicher kein Gewinn zu erzielen.

Daher wurde zur Ermittlung von annähernd realistischen Werten eine sehr simple Simulation des Pipeline- und Buszustands des Prozessors implementiert, mit der in Abhängigkeit von der Position einer Instruktion innerhalb des Blocks der Gewinn für jedes Datenobjekt berechnet werden kann. Dazu ist eine genaue Kenntnis über den Ablauf eines Datenzugriffs auf den SRAM beim TC1796 erforderlich. Leider werden darüber in den zahlreichen Dokumenten, die Infineon zur TriCore-Architektur veröffentlicht hat, keine allzu genauen Angaben gemacht (in [Inf04a] befindet sich lediglich eine unkommentierte Auflistung der Schritte für einem lesenden Zugriff). In der Pipeline-Analyse von aiT wird das Verhalten des TC1796 jedoch berücksichtigt, so dass durch Untersuchung der Ausgabedaten von aiT und Nachfrage bei dessen Hersteller AbsInt genügend Informationen über den Ablauf eines Datenzugriffs gewonnen werden konnten.

Als Überblick sind die an einem Datenzugriff beteiligten Komponenten des TC1796 in Abbildung 5.10 dargestellt. Während ein Zugriff auf den Scratchpad-Speicher durch das eng an die CPU gekoppelte *Data Memory Interface* durchgeführt wird, sind an einem SRAM-Zugriff weitere Komponenten beteiligt. Der Speicherzugriff wird von der CPU zunächst beim DMI angefordert, welches ihn dann über den Datenbus (DLMB) an die *Data Memory Unit* delegiert. Bei einem Lesezugriff gelangt der gelesene Wert dann über den selben Weg zurück zur CPU. Bei Schreibzugriffen wird von der DMU lediglich eine Bestätigung zurück zum DMI gesendet.

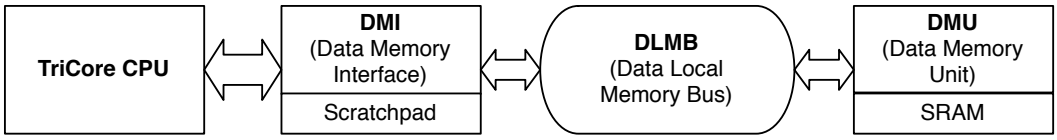

Abbildung 5.10: An einem SRAM-Zugriff beteiligte Komponenten des TC1796

Der TriCore Befehlssatz stellt Instruktionen zum Laden bzw. Speichern von 8, 16, 32 und 64-Bit Werten zur Verfügung. Diese unterscheiden sich jedoch nur aus Sicht der CPU, vom DMI aus werden immer 32-Bit Werte geschrieben bzw. gelesen. Daher verhält sich ein 64-Bit Zugriff für den Speicher genau wie zwei aufeinanderfolgende 32-Bit Zugriffe. Dennoch kann durch die Verwendung eines 64-Bit Zugriffs ein Zyklus gegenüber der Verwendung zweier 32-Bit Zugriffe gespart werden, da so nur eine Instruktion die Pipeline durchlaufen muss. 8 und 16-Bit Zugriffe zeigen exakt das gleiche Verhalten wie 32-Bit Zugriffe, der Inhalt des geschriebenen bzw. gelesenen Registers wird von der CPU nach dem Zugriff entsprechend beschnitten.

Ein lesender Zugriff auf den SRAM des TC1796 durchläuft dabei folgende Schritte, wobei jeder Schritt genau einen Taktzyklus benötigt:

1. Der Load-Befehl betritt die Instruction Fetch Phase der Pipeline.
2. In der Decode Phase wird die Operation dekodiert und der Wert des Adressregisters geholt.
3. In der Execute Phase wird die absolute Adresse des Zugriffs berechnet. Da die Writeback Phase erst betreten werden kann, wenn das zu ladende Datum vorliegt, wird die Pipeline nun gestoppt.
4. Das DMI fordert Zugriff auf den DLMB an.
5. Die Zugriffsadresse wird auf den Datenbus gelegt.
6. Nun erfolgt der Datenzugriff auf den SRAM innerhalb der DMU. Dieser kann innerhalb eines Taktes durchgeführt werden, und das angeforderte Datum wird auf den Datenbus gelegt.
7. Das DMI erhält die Antwort der DMU und speichert das Datum intern zwischen.
8. Die Daten werden nun vom DMI der CPU übergeben. Damit ist der Datenzugriff beendet und die Pipeline kann fortgesetzt werden.
9. In der Writeback-Phase wird der geladene Wert in das Zielregister geschrieben. Durch Zwischenpuffer in der Pipeline steht es aber auch für die anderen Instruktionen in der Pipeline bereits zur Verfügung.

Bei einem Zugriff auf den Scratchpad-Speicher könnte der Speicherzugriff in der Execute Phase direkt erfolgen und würde somit in der nächsten Pipeline-Stufe (Writeback) zur Verfügung stehen, deshalb treten in diesem Fall keine Wartezyklen auf (die Schritte 4 bis 8 werden nicht ausgeführt). Bei einem Zugriff auf den SRAM muss auf jeden Fall gewartet werden, bis das geladene Datum vorliegt, daher muss die Pipeline angehalten werden. Dies bedeutet auch, dass lesende Zugriffe immer komplett behandelt werden und danach sowohl DMI als auch DMU frei für weitere Anfragen sind. Der zu veranschlagende Gewinn durch Auslagerung des Datums in den Scratchpad-Speicher beträgt also mindestens 5 Zyklen.

Es kann vorkommen, dass das DMI durch einen Schreibzugriff aus einem vorherigen Befehl belegt ist und den Ladezugriff daher nicht sofort durchführen kann. In diesem Fall muss zunächst der Schreibzugriff beendet werden, was weitere Wartezyklen vor dem Übergang in Schritt 4 verursacht, die ebenfalls berücksichtigt werden müssen.

Ein schreibender Zugriff auf den SRAM verläuft ähnlich wie ein lesender, allerdings muss hier nicht auf Daten gewartet werden, daher ist ein Pipeline-Stall nur dann notwendig, wenn das DMI noch mit einem anderen Zugriff beschäftigt ist. Der Ablauf gestaltet sich daher wie folgt:

1. Der Store-Befehl betritt die Instruction Fetch Phase der Pipeline.
2. In der Decode Phase wird die Operation dekodiert und der Wert des Adressregisters geholt.

3. Nach der Berechnung der Zugriffsadresse in der Execute Phase kann diese vom DMI angefordert werden, falls dieses nicht noch auf einen Buszugriff wartet. Ansonsten wird die Pipeline angehalten, bis das DMI wieder Anforderungen annimmt.

4. Der Zugriff auf den Datenbus wird angefordert. Es kann sein, dass dieser noch durch einen der drei folgenden Schritte eines früheren Zugriffs belegt ist. Dann verbleibt das DMI so lange in diesem Status, bis der Bus frei ist. Die Pipeline kann währenddessen weiter bearbeitet werden, falls kein weiterer Zugriff auf das DMI benötigt wird.

5. Adresse und Wert des zu schreibenden Datums werden auf den Datenbus gelegt. Im gleichen Schritt wird das DMI wieder frei für eine weitere Anfrage.

6. Nun erfolgt der Schreibzugriff durch die DMU. Dieser wird ebenfalls innerhalb eines Taktes durchgeführt.

7. Die DMU legt eine Bestätigung des erfolgreichen Zugriffs auf den Bus, den das DMI empfängt.

Bei einem Schreibzugriff auf den Speicher hat die Writeback-Phase keine weiteren Aufgaben mehr zu erfüllen, daher ist diese hier nicht aufgeführt. Wie eine Abfolge von 3 Schreibzugriffen abläuft, ist in Abbildung 5.11 abgebildet. Neben den Phasen der Pipeline ist grau hinterlegt der Zustand von DMI und DMU aufgeführt. Die Phasen *BUS ACCESS*, *DMI STORE*, *DMU STORE* und *ACK* entsprechen dabei den Punkten 4 bis 7 in der oberen Auflistung.

	STORE 1		STORE 2		STORE 3	
1	FETCH					
2	DECODE		FETCH			
3	EXEC		DECODE		FETCH	
4	WRITEBACK	BUS ACCESS	EXEC		DECODE	
5		DMI STORE	WRITEBACK	BUS ACCESS	EXEC	
6		DMU STORE		BUS ACCESS	STALL	
7		ACK		BUS ACCESS	STALL	
8				DMI STORE	WRITEBACK	BUS ACCESS
9				DMU STORE		BUS ACCESS
10				ACK		BUS ACCESS
11						DMI STORE
12						DMU STORE
13						ACK

Abbildung 5.11: Pipeline- und DMI-Zustand bei drei aufeinanderfolgenden Schreibzugriffen

In diesem Beispiel sind die zwei unterschiedlichen Arten von Wartezuständen ersichtlich. In Store 2, Takt 6-7 wird vom DMI auf das Freiwerden des Busses gewertet. Dies verursacht keinen Pipeline-Stall, da der Store Befehl aus Sicht der CPU vollständig ausgeführt wurde. Ein Stall

tritt erst bei Store 3 auf, weil dort das DMI noch in der "Bus Access" Phase verharrt und keine weiteren Anforderungen der CPU annimmt. Daher muss hier die Pipeline angehalten werden, bis das DMI seine Schreibanfrage auf den Bus legen kann (Takt 8).

Eine Simulation des hier beschriebenen Verhaltens wurde für die Berechnung der Blockgewinne implementiert. Dabei wird als zu erwartender Gewinn durch eine Auslagerung die Summe der Wartezyklen innerhalb der Pipeline bestimmt, da diese bei einem Zugriff auf den Scratchpad-Speicher nicht auftreten würden.

Werden im Quelltext eines Programms Variablen als `const` deklariert, so kann deren Wert nicht verändert werden. Durch den WCC werden diese dann normalerweise im Datenflash des TC1796 gespeichert. Leider sind auch für diesen keine Angaben zu Zugriffszeiten verfügbar. Da aber auf solche Datenobjekte nur lesend zugegriffen werden kann und lesende Zugriffe immer die Pipeline stoppen, bis das Datum verfügbar ist, konnte der Gewinn der Auslagerung eines konstanten Datums durch Messungen mit aiT bestimmt werden. Er beträgt 16 Zyklen.

Einschränkungen

Auf die oben beschriebene Weise können natürlich keine absolut genauen Werte berechnet werden. Mehrere Einschränkungen verhindern dies:

- Es wird nur die Load/Store Pipeline simuliert. Ein Stall auf der Integer-Pipeline oder die Ausführung von Instruktionen, die mehr als einen Zyklus benötigen, kann somit nicht berücksichtigt werden. Letztere sind allerdings im TriCore Befehlssatz extrem selten.
- Jeder Basisblock wird für sich betrachtet. Der Pipelinezustand aus den Vorgängern des Blocks ist somit nicht bekannt. Wenn also die letzte Instruktion eines Vorgängers einen Stall verursacht, kann dieser im aktuellen Block nicht mehr berücksichtigt werden.
- Die Auslagerung eines Datenobjekts kann Einfluss auf alle weiteren Zugriffe haben, da ja durch diese der Pipelinezustand verändert wird. Da jedoch immer nur der durch die Auslagerung eines Datenobjekts erzielte Gewinn ermittelt wird, ist hier nur diese isolierte Betrachtung möglich. Es kann aber durchaus sein, dass bei gemeinsamer Auslagerung zweier Datenobjekte in einem Block der tatsächliche Gewinn von der Summe der einzeln berechneten Gewinne abweicht. Dies ist die bedeutendste Einschränkung und erklärt zugleich, weshalb eine genauere Ermittlung der Gewinne nicht möglich ist. Deshalb wäre auch die Implementierung eines noch detaillierteren Pipelinemodells, das die beiden vorhergehenden Punkte abdeckt, nicht lohnenswert.

Auch, wenn auf diese Weise keine exakten Werte ermittelt werden, so ist die Abschätzung aber genau genug, um innerhalb des ILP eine brauchbare Gewinnabschätzung zu liefern.

5.4.3 Lösung des ILP

Die Lösung des ILP erfolgt durch einen externen LP-Solver (wahlweise `lp_solve` [lps08] oder `cplex` [cpl08]). Dieser wird durch die bereits erwähnte Bibliothek *libilp* aufgerufen, die daraufhin die Lösung zurück in die objektorientierte Beschreibung des ILP überträgt. Die Werte der Variablen und der Zielfunktion stehen dann direkt zur Verfügung.

Aus den Entscheidungsvariablen kann so der Zielspeicher für jedes Datenobjekt bestimmt werden. Falls eine Auslagerung in den Scratchpad bestimmt wurde (erkennbar an dem Variablenwert 1), wird das LLIR-Element der entsprechenden Sektion zugeordnet, so dass es später durch den Linker dort gespeichert wird. Der Inhalt des Scratchpad-Speichers ist damit definiert. Für das so modifizierte Programm ändern sich nur die Adressen der Datenobjekte, die erst später durch den Linker bestimmt werden. Es ist also keine Veränderung des Kontrollflusses oder ein Einfügen von Instruktionen nötig. Daher ist die Umsetzung der einmal ermittelten Scratchpad-Allokation sehr simpel.

Ein Nachteil ist jedoch, dass der Scratchpad-Speicher eventuell nicht so effizient wie möglich ausgenutzt werden kann, da ein dort gespeichertes Element diesen nicht wieder verlässt und somit auch dann darin verbleibt, wenn gar nicht mehr darauf zugegriffen wird. Im nächsten Kapitel soll daher die Möglichkeit untersucht werden, den Inhalt des Scratchpad-Speichers während der Laufzeit des Programms zu ändern.

Kapitel 6

Dynamische Scratchpad-Allokation

Ein Algorithmus zur Scratchpad-Allokation wird als dynamisch bezeichnet, wenn sich der Inhalt des Scratchpad-Speichers zur Laufzeit des Programms ändern kann. Im Gegensatz zu einer statischen Allokation kann so der schnellere Speicher unter Umständen effizienter genutzt werden. Dies liegt daran, dass Programme häufig lokal agieren. In unterschiedlichen Programmteilen wird dann auf unterschiedliche Mengen von Datenobjekten zugegriffen. Durch eine dynamische Allokation kann dafür gesorgt werden, dass sich immer genau die gerade benötigten Daten im schnelleren Speicher befinden. Dies geschieht durch die Ein- und Auslagerung der Datenobjekte zur Laufzeit des Programms, wozu zusätzlicher Code notwendig ist. Da auch dieser so genannte *Spillcode* zur Laufzeit des Programms beiträgt, muss abgewogen werden, an welchen Stellen sich eine Auslagerung lohnt.

Dieses Kapitel beschreibt die Erweiterung des bisher vorgestellten Algorithmus, so dass eine dynamische Allokation vorgenommen wird. Dazu wird zunächst auf die Problemstellung und die Anforderungen an diese Erweiterung eingegangen. Daraufhin werden die Änderungen, die an der ILP-Formulierung aus Kapitel 4 nötig sind, erläutert. Die Umsetzung der dynamischen Scratchpad-Allokation im WCC wird im dritten Abschnitt dieses Kapitels beschrieben. Schließlich wird auf die Einschränkungen des Algorithmus bzw. der aktuellen Implementierung eingegangen.

Das vorgestellte ILP-Verfahren wurde ausgehend von dem in Kapitel 4 vorgestellten Verfahren neu entwickelt. Es existiert bisher kein vergleichbarer Ansatz, der eine dynamische Scratchpad-Allokation zur WCET-Optimierung anhand eines verschachtelten Kontrollflussgraphen bei gleichzeitiger impliziter Betrachtung aller möglichen Kontrollflusspfade vornimmt. Aus dieser Form resultieren einige Vorteile, wie zum Beispiel die vereinfachte Berechnung von Speicheradressen für in den Scratchpad-Speicher ausgelagerte Datenobjekte. Allerdings ergeben sich so auch einige Einschränkungen, auf die ebenfalls innerhalb dieses Kapitels eingegangen wird.

6.1 Problemstellung

Der bisher vorgestellte Algorithmus berechnet eine statische Scratchpad-Allokation. Dazu benutzt er eine spezielle Darstellung des Kontrollflussgraphen, in denen Schleifen und Funktionen als Knoten mit einem Untergraphen behandelt werden. Es soll nun untersucht werden, inwiefern sich diese Darstellung nutzen lässt, um eine dynamische Allokation zu berechnen. Dazu soll, un-

ter Berücksichtigung der Kosten des nötigen Spillcodes, für jeden Knoten eine günstige Belegung des Scratchpad-Speichers gewählt werden. Dies erfordert die Kenntnis, welche Datenobjekte vor dem Betreten bzw. nach dem Verlassen eines Knotens ein- oder ausgelagert werden müssen.

Andere Arbeiten über dynamische Scratchpad-Allokation - sei es zur Optimierung von ACET, WCET oder Energieverbrauch - berechnen diese Informationen meist anhand der Kanten im Kontrollflussgraphen (siehe z.B. [UB03, UDB06, DP07, SWLM02]). Eine Aus- bzw. Einlagerung von Daten ist so potentiell an jeder Kante möglich. Dieser Ansatz ist mit dem vorgestellten Konzept der Schleifenkontraktion nicht vereinbar, da in dieser Darstellung nicht mehr alle Kanten des Graphen explizit vorhanden sind.

Es kann jedoch davon ausgegangen werden, dass eine Änderung des Scratchpad-Inhalts fast ausschließlich vor bzw. nach Schleifen oder Funktionsaufrufen sinnvoll ist. Dies liegt zum einen daran, dass sich Programme häufig lokal verhalten und daher innerhalb einer Funktion jeweils auf eine bestimmte Menge von Datenobjekten zugreifen. Zum anderen ist für das dynamische Auslagern von Datenobjekten Spillcode erforderlich. Der durch eine Auslagerung erzielte Gewinn ist nur dann höher als die durch den Spillcode verursachten Kosten, wenn sehr häufig auf das ausgelagerte Objekt zugegriffen wird. Dies ist typischerweise genau innerhalb von Schleifen der Fall.

Sowohl Schleifen als auch Funktionen sind als Knoten in der in dieser Arbeit verwendeten Form des Kontrollflussgraphen vorhanden, und daher soll genau hier eine Auslagerung erfolgen können.

6.2 ILP-Formulierung

Das Problem ist nun, die Veränderlichkeit des Scratchpad-Inhalts in der ILP-Formulierung auszudrücken. Dazu müssen mehrere Punkte beachtet werden:

- Die Scratchpad-Belegung ist nicht mehr global für das gesamte Programm, sondern pro Schleifen- bzw. Funktionsknoten zu berechnen.
- Bei der Bestimmung der Blockkosten muss die Information verfügbar sein, welche Variablen sich innerhalb eines Blocks im Scratchpad-Speicher befinden.
- Die Kosten des Spillcodes, der nötig ist, um Datenobjekte ein- bzw. auszulagern, müssen im ILP berücksichtigt werden.
- Der Kontrollfluss soll weiterhin durch die gleichen Constraints wie zur statischen Allokation beschrieben werden.
- Auch die Minimierung der WCET soll durch das gleiche Prinzip wie zuvor erfolgen.

6.2.1 Anpassung der Entscheidungsvariablen

Um das ILP entsprechend anzupassen, muss zunächst den Entscheidungsvariablen eine zweite Dimension hinzugefügt werden. Anstatt einer Variablen pro Datenobjekt, die dessen Speicherort für das gesamte Programm codiert, muss nun für jeden Schleifen- oder Funktionsknoten

gespeichert werden, welche Datenobjekte sich innerhalb des Schleifenkörpers bzw. der Funktion im Scratchpad-Speicher befinden. Statt der Variablen S_d werden nun also die Variablen $S_{n,d}$ definiert. Diese haben folgende Bedeutung:

$$S_{n,d} = \begin{cases} 1 & \text{falls } d \text{ innerhalb des Knotens } n \text{ im Scratchpad liegt} \\ 0 & \text{sonst} \end{cases} \tag{6.1}$$

An dieser Stelle muss noch erläutert werden, was genau "innerhalb des Knotens n" bedeutet. Bei der Betrachtung verschachtelter Schleifen ist dies nämlich nicht eindeutig. Dies soll anhand des Beispiels in Abbildung 6.1 erläutert werden. Hier existieren drei Knoten, für die eine unterschiedliche Scratchpad-Belegung möglich ist: *main*, *loop_b* und *loop_c*. Enthielte das Programm zwei Datenobjekte X und Y, so würden innerhalb des ILP folgende sechs Entscheidungsvariablen definiert: $S_{main,X}$, $S_{main,Y}$, $S_{loop\_b,X}$, $S_{loop\_b,Y}$, $S_{loop\_c,X}$ und $S_{loop\_c,Y}$.

Ein Basisblock gehört immer zu dem nächsthöheren Graphen. Obwohl beispielsweise der Knoten c auch Teil der Funktion *main* und der Schleife *loop_b* ist, gehört er zu dem Untergraphen von *loop_c*. Dies ist daher der Knoten, durch dessen Entscheidungsvariablen der Speicherort von den in c referenzierten Variablen bestimmt wird.

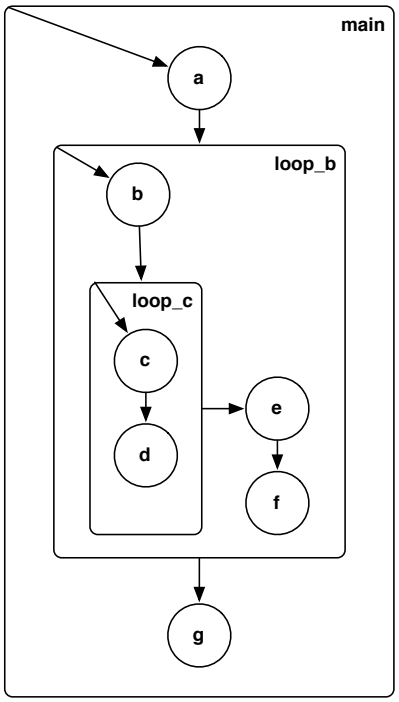

Abbildung 6.1: Verschachtelter Graph mit 2 Schleifen

Somit wird für jeden verschachtelten Knoten eine eigene Scratchpad-Belegung gewählt, die für alle darin direkt enthaltenen Basisblöcke gilt. Dies ist gerade für Schleifen und Funktionen sinnvoll, da innerhalb von diesen häufig auf eine ganz bestimmte Menge von Datenobjekten zugegriffen wird. Die Entscheidung, ob sich die Belegung beim Betreten einer tieferen Verschachtelungsebene ändert, wird so ebenfalls durch das ILP getroffen.

Die Kapazitätsbeschränkung des Scratchpad-Speichers kann nun nicht mehr global ausgedrückt werden. Daher wird für jeden Knoten, der einen Untergraphen besitzt, eine eigene Restriktion benötigt:

$$\forall n \in collapsed\_nodes: \quad \sum_{d \in data\_objects} S_{n,d} * size_d \leq scratchpad\_size \tag{6.2}$$

Nun ist es notwendig, die so ausgedrückte Scratchpad-Belegung auch innerhalb der Blockkosten-Constraints zu berücksichtigen. Für jeden Basisblock ist der Knoten bekannt, in dessen Untergraphen er sich befindet. Für Blöcke innerhalb von Schleifen ist dies der nächsthöhere Schleifenknoten, für Blöcke außerhalb von Schleifen der jeweilige Funktionsknoten. In der folgenden Restriktion wird dieser Knoten als $parent(b)$ bezeichnet.

$$\forall b \in basic\_blocks: \quad C_b = wcet_b - \sum_{d \in data\_objects} S_{parent(b),d} * gain_{b,d} \tag{6.3}$$

Im Gegensatz zu den in Kapitel 4.4.2 beschriebenen Constraints werden die Kosten eines eventuellen Funktionsaufrufs innerhalb des Basisblocks b an dieser Stelle nicht addiert. Der Grund hierfür wird im nächsten Abschnitt erläutert.

6.2.2 Berücksichtigung der Kopierkosten

Das so abgeänderte ILP würde bereits jetzt für jeden Knoten die günstigste Belegung berechnen, allerdings wird bisher nicht berücksichtigt, dass das Kopieren der Datenobjekte in den bzw. aus dem Scratchpad-Speicher Kosten verursacht, die den lokalen Gewinn durch eine Auslagerung übertreffen können. Um diese berücksichtigen zu können, muss bekannt sein, an welchen Stellen ein Kopieren notwendig ist. Dies muss ebenfalls durch Variablen und Constraints abgebildet werden.

Zu diesem Zweck werden die Variablen $Load_{n,d}$ und $Store_{n,d}$ eingeführt, mit deren Hilfe gekennzeichnet wird, ob ein Datenobjekt d sich innerhalb des Knotens n in einem anderen Speicher befindet als im Vaterknoten von n. Ein solcher Fall würde auf jeden Fall das Kopieren des entsprechenden Datenobjekts vor dem Betreten von n erfordern. Ob auch ein Zurückkopieren in den ursprünglichen Speicher nach Verlassen des Knotens n erforderlich ist, hängt davon ab, ob d innerhalb von n überhaupt verändert werden kann. Dies wird erst später innerhalb der Kopierkosten berücksichtigt und braucht durch die Variablen nicht ausgedrückt werden. Diese haben daher folgende Bedeutung:

$$Load_{n,d} = \begin{cases} 1 & \text{falls } d \text{ innerhalb des Knotens } n \text{ im Scratchpad-Speicher} \\ & \text{liegt, in } parent(n) \text{ jedoch nicht} \\ 0 & \text{sonst} \end{cases} \tag{6.4}$$

$$Store_{n,d} = \begin{cases} 1 & \text{falls } d \text{ innerhalb des Knotens } parent(n) \text{ im} \\ & \text{Scratchpad-Speicher liegt, in } n \text{ jedoch nicht} \\ 0 & \text{sonst} \end{cases} \tag{6.5}$$

Diese Variablen werden für alle Knoten definiert, an denen sich die Belegung des Scratchpad-Speichers ändern kann. Ihre Belegung wird durch Constraints aus den Entscheidungsvariablen der entsprechenden Knoten abgeleitet. Für Schleifenknoten ist dies ohne Probleme möglich, da für diese der Untergraph, in dem sie enthalten sind, bekannt ist. Somit besitzt jeder Schleifenknoten einen eindeutigen Vaterknoten.

Funktionen hingegen können von unterschiedlichen Stellen des Programms aus aufgerufen werden. An den Funktionsknoten selbst ist daher die Variablenbelegung des aufrufenden Knotens nicht bekannt. Dieses Problem kann behandelt werden, indem die *Load*/*Store* Variablen statt für die Funktion selbst für einen die Funktion aufrufenden Dummyknoten definiert werden, der lediglich den Funktionsaufruf selbst enthält. Vor und nach diesem ist auch das Einfügen des Spillcodes notwendig, der den für die aufgerufene Funktion gewählten Scratchpad-Inhalt erzeugt bzw. den ursprünglichen Zustand wiederherstellt.

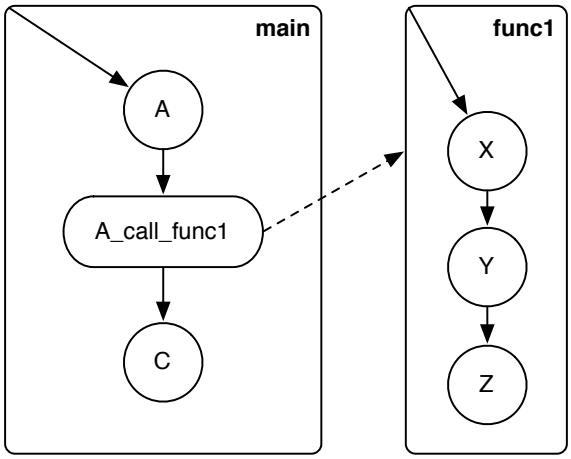

Abbildung 6.2: Nutzung eines Dummyknotens zur eindeutigen Positionierung von Kopierkosten

Die Erstellung eines solchen Dummyknotens wurde bereits in Kapitel 5 im Rahmen der Beschreibung der Datenstruktur des Kontrollflussgraphen erläutert. Dieser Mechanismus wird nun auch hier verwendet, um eine eindeutige Position für das Kopieren von Daten zwischen Scratchpad und SRAM zu schaffen. In dem in Abbildung 6.2 dargestellten Beispiel wurde so für den Aufruf von $func1$ aus dem Basisblock A ein weiter Knoten (A_call_func1) eingefügt. Für diesen werden die *Load*/*Store* Variablen erzeugt, obwohl die Entscheidungsvariablen für die Scratchpad-Belegung ($S_{n,d}$) für die Funktion $func1$ selbst definiert sind.

Damit dieses Verfahren funktioniert, muss die Belegung der Entscheidungsvariablen innerhalb des Dummyknotens mit deren Belegung innerhalb der aufgerufenen Funktion übereinstimmen. Es ist jedoch nicht nötig, dies durch zusätzliche Variablen auszudrücken. Bei der Erzeugung des ILP müssen lediglich für jeden Dummyknoten die Variablen der aufgerufenen Funktion genutzt werden. Dies ist auch in den folgenden Constraints zu beachten: Wenn dort $S_{n,d}$ genutzt wird, ist für solche Dummyknoten die entsprechende Variable des Knotens der aufgerufenen Funktion gemeint.

Damit das ILP weiterhin selbstständig die optimale Scratchpad-Allokation berechnet, muss die Belegung der *Load*/*Store* Variablen ausschließlich durch die Entscheidungsvariablen $S_{n,d}$ definiert werden. Das Laden eines Datenobjekts in den Scratchpad-Speicher ist nur dann erforderlich,

$S_{parent(n),d}$	$S_{n,d}$	$Load_{n,d}$	6.6	6.7	6.8
0	0	0	✓	✓	✓
0	0	1	✓	✓	×
0	1	0	×	✓	✓
0	1	1	✓	✓	✓
1	0	0	✓	✓	✓
1	0	1	✓	×	×
1	1	0	✓	✓	✓
1	1	1	✓	×	✓

Tabelle 6.1: Beweis der Korrektheit der Load-Constraints

wenn es im übergeordneten Knoten nicht ausgelagert ist, in dem aktuellen jedoch schon. In der ILP-Formulierung kann dies durch jeweils drei Constraints ausgedrückt werden:

$$S_{n,d} - S_{parent(n),d} - Load_{n,d} \leq 0 \tag{6.6}$$

$$S_{parent(n),d} + Load_{n,d} \leq 1 \tag{6.7}$$

$$Load_{n,d} - S_{n,d} \leq 0 \tag{6.8}$$

Die erste Restriktion sorgt dafür, dass $Load_{n,d}$ den Wert 1 erhält, wenn das Datenobjekt d im aktuellen Knoten n im Scratchpad-Speicher liegt, in dessen Vaterknoten jedoch nicht. Durch die zweite Restriktion wird sichergestellt, dass ein Laden niemals notwendig ist, falls das Datenobjekt schon im Vaterknoten ausgelagert ist. Constraint 6.8 schließlich schließt aus, dass eine Variable geladen wird, obwohl gar keine Auslagerung für den aktuellen Knoten bestimmt wurde.

Die Korrektheit dieser Constraints lässt sich anhand einer Wahrheitstabelle zeigen. Dazu wird jede mögliche Belegung der drei betroffenen Variablen aufgeführt und überprüft, ob diese durch mindestens eine der drei Restriktionen ausgeschlossen wird. Somit wird deutlich, dass innerhalb des ILP nur die Belegungen möglich sind, die einer korrekten Definition der *Load* Variablen entsprechen (s. Tabelle 6.1).

Analog lassen sich solche Constraints auch für die Variablen $Store_{n,d}$ definieren. Auch der Beweis der Korrektheit kann auf die gleiche Art und Weise erfolgen, soll hier aber nicht geführt werden.

$$S_{parent(n),d} - S_{n,d} - Store_{n,d} \leq 0 \tag{6.9}$$

$$S_{n,d} + Store_{n,d} \leq 1 \tag{6.10}$$

$$Store_{n,d} - S_{parent(n),d} \leq 0 \tag{6.11}$$

Zur Unterstützung des LP-Solvers kann dann noch folgende Restriktion definiert werden, welche ausdrückt, dass in jedem Knoten pro Datenobjekt nur eine Ein- oder Auslagerung erfolgen kann.

$$Load_{n,d} + Store_{n,d} \leq 1 \tag{6.12}$$

Nun, da für jeden Knoten bekannt ist, welche Datenobjekte kopiert werden müssen, können die entsprechenden Kosten in den Knoten berücksichtigt werden. Dies geschieht sowohl für Schleifenknoten als auch für die Knoten, die einen Funktionsaufruf darstellen.

Dazu ist die Kenntnis der Anzahl von Zyklen erforderlich, die durch das Ein- und Auslagern der Datenobjekte verursacht werden. Diese können abhängig von dem aktuellen Knoten des Graphen sein und werden daher als $loadcost_{n,d}$ und $storecost_{n,d}$ bezeichnet. Insbesondere muss in diesen Kosten berücksichtigt werden, ob nach Verlassen des Knotens n ein Zurückkopieren von d in den ursprünglichen Speicher notwendig ist. Prinzipiell muss dies immer geschehen, da innerhalb der Basisblöcke im umliegenden Graphen davon ausgegangen wird, dass d in dem entsprechenden Speicher liegt. Es gibt jedoch eine Ausnahme: Ein Zurückkopieren eines Datenobjekts d aus dem Scratchpad-Speicher ist nicht notwendig, wenn d innerhalb des Knotens n (inklusive aller tieferen Verschachtelungsebenen) nicht geändert wurde. Dies umschließt natürlich insbesondere den Fall, dass d als `const` deklariert wurde und somit gar nicht änderbar ist.

Die Kosten, die das eigentliche Kopieren eines Datenobjekts zwischen den Speichern verursacht, können relativ genau abgeschätzt werden, da sie lediglich von der Größe des Objekts abhängig sind. Genauer wird hierauf im Abschnitt 6.3.4 eingegangen.

Es wäre an dieser Stelle möglich, auch weitere Kopiervorgänge nicht zu berücksichtigen. Ein geändertes Datenobjekt, das im restlichen Programmverlauf nicht mehr genutzt wird, müsste so ebenfalls nicht zurück kopiert werden. Diese Annahme ist jedoch in eingebetteten Systemen gefährlich, da die Ausgabe eines Algorithmus häufig Eingabe einer weiteren Komponente innerhalb des Systems ist. Diese wird dann an einer bestimmten Speicherstelle erwartet, weswegen ein Zurückkopieren geänderter Daten in jedem Fall erfolgen sollte.

Die Kosten einer Schleife werden nun wie folgt definiert:

$$\begin{aligned} C_{loop} = \quad & W_{first\_node(loop)} * wcec_{first\_node(loop)} \\ & + \sum_{d \in data\_objects} (Store_{loop,d} * storecost_{loop,d}) \\ & + \sum_{d \in data\_objects} (Load_{loop,d} * loadcost_{loop,d}) \end{aligned} \tag{6.13}$$

Da Funktionsaufrufe nun eigene Knoten darstellen, werden an dieser Stelle neben den Kopierkosten auch die Kosten des Funktionsaufrufs selbst angerechnet (anstatt wie zuvor innerhalb der Blöcke, die den Aufruf beinhalten). Die Behandlung rekursiver Aufrufe erfolgt dabei wie bereits in Kapitel 4 beschrieben. Die zu erzeugenden Constraints für Funktionsaufrufe haben damit folgende Form:

$$\begin{aligned} C_{call} = \quad & recursion\_depth(call) * C_{called\_func(call)} \\ & + \sum_{d \in data\_objects} (Store_{call,d} * storecost_{call,d}) \\ & + \sum_{d \in data\_objects} (Load_{call,d} * loadcost_{call,d}) \end{aligned} \tag{6.14}$$

6.2.3 Abbildung des Kontrollflusses

Die restlichen Constraints zur Abbildung des Kontrollflusses, also insbesondere zur Bestimmung der Knotengewichte, ändern sich gegenüber der in Kapitel 4 beschriebenen Formulierung nicht und können so übernommen werden. Auch die Zielfunktion kann daher unverändert bleiben und sorgt für die Minimierung des Gewichtes im Startknoten. Somit wird weiterhin die WCET des Programms minimiert, es bestehen allerdings zusätzliche Freiheiten durch die gestiegene Anzahl an Variablen.

Damit ist ein ILP formuliert, durch dessen Lösung für jeden verschachtelten Knoten innerhalb des Kontrollflussgraphen eine Variablenbelegung bestimmt wird. Daraus lässt sich leicht die entsprechende Variablenbelegung für jeden Basisblock des Graphen ableiten, so dass auch hier zur Übersetzungszeit der Inhalt des Scratchpad-Speichers an jeder Programmstelle bekannt ist.

6.3 Umsetzung

Neben den beschriebenen Änderungen der ILP-Formulierung ist für die Implementierung der dynamischen Allokation noch weiterer Aufwand zu betreiben. Zum einen müssen die tatsächlichen Speicheradressen für jedes Element innerhalb der verschachtelten Knoten berechnet werden. Des Weiteren ist natürlich die Erzeugung von Spillcode und dessen Einfügen an den richtigen Stellen im Programm nötig.

Die ersten Schritte des in Kapitel 5 für die statische Allokation beschriebenen Vorgehens (insbesondere der Aufbau des Kontrollflussgraphen) müssen für die dynamische Allokation nicht geändert werden. Erst bei der Formulierung des ILP und dem Anwenden der Lösung sind Anpassungen nötig. Abbildung 6.3 zeigt, welche Schritte nötig sind, um eine dynamische Auslagerung der Datenobjekte umzusetzen. Die abgebildeten Punkte werden in den folgenden Abschnitten erläutert.

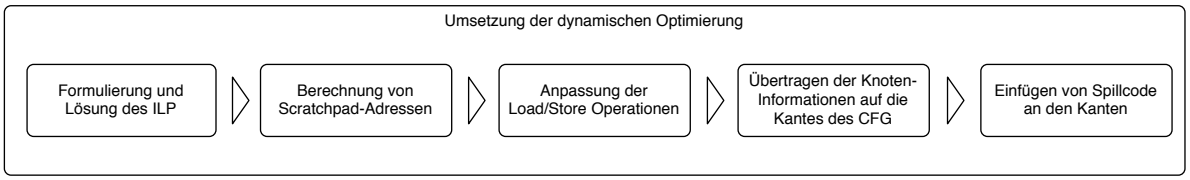

Abbildung 6.3: Umsetzung der dynamischen Optimierung

6.3.1 Anpassung der ILP-Formulierung

Zunächst soll betrachtet werden, welche Änderungen sich für die Implementierung der ILP-Formulierung ergeben. Da die Datenstruktur zur Beschreibung des Kontrollflussgraphen von vornherein darauf ausgelegt wurde, universell einsetzbar zu sein, kann dieser wie bereits beschrieben aufgebaut werden. Auch die Vorgehensweise des Algorithmus, den Graphen beginnend bei der Startfunktion rekursiv zu durchlaufen, kann beibehalten werden. Lediglich die Constraints, die dabei erzeugt werden, müssen abgeändert oder erweitert werden:

- **Kapazitätsbeschränkungen**
  Die Beschränkung der Scratchpad-Größe muss nun für jeden verschachtelten Knoten modelliert werden. Da diese Knoten alle von der Klasse *CollapsedNode* erben, können die entsprechenden Constraints genau hier erzeugt werden.
- **Kopierkosten bei Schleifen**
  Für jede Schleife müssen nun zusätzlich die durch den benötigten Spillcode auftretenden Kosten berücksichtigt werden. Die in Abschnitt 6.2.2 beschriebenen Constraints werden also an jedem Knoten vom Typ *LoopNode* erzeugt.
- **Kopierkosten bei Funktionsaufrufen**
  Dasselbe gilt für Funktionsaufrufe. Auch hier müssen Kopierkosten berücksichtigt werden, daher werden auch Knoten vom Typ *FunctionCallNode* entsprechend behandelt.

Zur Formulierung der entsprechenden Constraints fehlt lediglich die Berechnung der Kopierkosten. Hierbei sind mehrere Punkte zu beachten. Zum einen sind die Kosten eines Kopiervorgangs abhängig vom Aufbau des verwendeten Spillcodes. Ist dieser bekannt, kann die dafür benötigte Laufzeit relativ genau aus der Größe des zu kopierenden Datenobjekts abgeschätzt werden. Die Erzeugung des Spillcodes wird in einem späteren Abschnitt erläutert. Zum anderen muss berücksichtigt werden, dass das Zurückkopieren eines Datenobjekts nicht notwendig ist, wenn gar nicht schreibend darauf zugegriffen wurde. Für konstante Variablen werden diese Kosten daher niemals angerechnet, für alle weiteren Objekte wird geprüft, ob innerhalb des entsprechenden Knotens ein Schreibzugriff darauf stattfinden kann.

Nach der Lösung des ILP steht für jeden verschachtelten Knoten eine Scratchpad-Belegung fest, die nun nur noch umgesetzt werden muss. Dazu müssen zunächst konkrete Adressen für die einzelnen Datenobjekte berechnet werden.

6.3.2 Berechnung von Speicheradressen

In dem durch die statische Optimierung berechneten Speicherlayout befindet sich jedes Datenobjekt für die gesamte Laufzeit entweder im Scratchpad oder im langsameren Speicher (SRAM bzw. Flash). Da sich bei einer dynamischen Optimierung der Scratchpad-Inhalt zur Laufzeit mehrmals ändern kann, behält jedes Objekt seine feste Adresse im langsameren Speicher. Diese wird bei Zugriffen von Programmstellen, an denen das Objekt nicht ausgelagert ist, verwendet.

Zusätzlich muss nun für alle ausgelagerten Objekte auch eine Adresse im Scratchpad-Speicher berechnet werden. Dieses Vorgehen nennt man *overlaying memory allocation*, da sich so im Scratchpad immer eine Überlagerung des restlichen Speichers befindet. Es ist vergleichbar mit

dem *demand paging* zwischen Hauptspeicher und Plattenlaufwerken, das Ein- und Auslagern erfolgt aber im Unterschied dazu bereits zur Übersetzungszeit und nicht erst zur Laufzeit des Programms. Dies sichert weiterhin die Vorhersagbarkeit des Programmverhaltens und ermöglicht somit eine enge WCET-Abschätzung.

Die Eigenschaft, dass eine Scratchpad-Belegung für jeden verschachteltem Knoten berechnet wurde, bringt hier einen großen Vorteil: Bei einer Berechnung auf Basis von Kanten ist es schwierig, ein Objekt an unterschiedlichen Speicheradressen zu sichern. In [VM06] wird daher für jedes ausgelagerte Objekt nur eine Adresse im Scratchpad-Speicher bestimmt, obwohl ein Kopieren in den und aus dem Speicher an unterschiedlichen Stellen möglich ist. Dies erhöht die Wahrscheinlichkeit, dass keine freie Adresse für ein auszulagerndes Objekt gefunden werden kann. Durch die verschachtelte Struktur des Kontrollflussgraphen kann im hier vorgestellten Ansatz leicht eine Unterstützung unterschiedlicher Scratchpad-Adressen für ein und dasselbe Objekt implementiert werden. Es ist lediglich darauf zu achten, dass in aufeinanderfolgenden Codesegmenten die gleiche Adresse verwendet wird, da sonst ein sehr kostspieliges Umkopieren notwendig würde.

Der Algorithmus zur Zuweisung von Speicheradressen durchläuft den Kontrollflussgraphen genau wie jener zur Formulierung des ILP rekursiv. Er wird nacheinander für jedes Datenobjekt aufgerufen, beginnt bei der Startfunktion und folgt auch Funktionsaufrufen. In jedem verschachtelten Knoten wird für das aktuelle Datenobjekt (falls es sich im Scratchpad-Speicher befinden soll) eine Adresse berechnet. Die Zuweisung von Adressen zu neu einzulagernden Objekten geschieht dabei per *first fit* Verfahren. Bereits eingelagerte Objekte behalten die Adresse, die sie auch im übergeordneten Knoten besitzen. Dieses Vorgehen wird in Algorithmus 1 verdeutlicht. Die Variable *data* speichert hier das momentan behandelte Datenobjekt, *node* stellt den aktuellen Knoten des Graphen dar und *previous* dessen Vorgänger. Für Schleifenknoten ist dies deren Vaterknoten, bei Funktionsaufrufen wird der aufrufende Knoten als Vorgänger übergeben. Da die Funktion *main* keinen Vorgänger besitzt, ist für sie eine Sonderbehandlung erforderlich, die allerdings für die Funktionsweise des Algorithmus unbedeutend ist und daher nicht aufgeführt wird.

Nachdem der Kontrollflussgraph auf diese Weise für jedes Datenobjekt einmal durchlaufen wurde, stehen alle Scratchpad-Adressen fest. Kann die Funktion *allocateFirstFit* aufgrund von Fragmentierung nicht genügend freien Platz am Stück finden, um ein Datenobjekt im Scratchpad unterzubringen, so kann dieses nicht eingelagert werden. Daher wird in einem solchen Fall keine Adresse zugewiesen. Dieses Problem wird im Abschnitt 6.4 genauer betrachtet.

6.3.3 Anpassen der Lade- und Speicherbefehle

Nun, da für alle Datenobjekte die Speicheradressen an jeder Stelle das Programms bekannt sind, müssen diese Änderungen innerhalb der LLIR umgesetzt werden. Die Zieladressen der Load/Store Instruktionen sind dafür je nach aktuellem Speicherort der Datenobjekte, auf die zugegriffen wird, entsprechend anzupassen. Befindet sich ein Datenobjekt innerhalb eines Basisblocks im Scratchpad-Speicher, so müssen alle Instruktionen, welche dieses Objekt referenzieren, entsprechend angepasst werden. Dies betrifft neben den Load/Store Befehlen, die einen Speicherzugriff durchführen, auch andere Operationen, die zu Adressberechnungen genutzt werden. In den Parametern dieser Befehle muss das Label des entsprechenden Datenobjekts durch die im vorigen Schritt berechnete Scratchpad-Adresse ersetzt werden.

Algorithmus 1 Zuweisung von Speicheradressen

```
1:  function processNode(data, node, previous)
2:      if (node ∈ visited) then return
3:      else
4:          visited ← (visited ∪ node)
5:      end if
6:      if (S_{node,data} == 0) then
7:          address_{node,data} ← NONE
8:      else if (address_{previous,data} == NONE) then
9:          address_{node,data} ← allocateFirstFit(node, data)
10:     else
11:         address_{node,data} ← address_{previous,data}
12:     end if
13:     if (type(node) == CollapsedNode) then
14:         for all (subnode ∈ subnodes(node)) do
15:             processNode(data, subnode, node)          ▷ rekursiver Aufruf
16:         end for
17:     else if (type(node) == FunctionCallNode) then
18:         processNode(data, calledFunction(node), node)  ▷ rekursiver Aufruf
19:     end if
20: end function
```

Dies wirft jedoch ein Problem auf, wenn Adressberechnungen nicht im selben Knoten wie der tatsächliche Zugriff stattfinden. In Abschnitt 6.4 wird dieses Problem genauer betrachtet.

6.3.4 Einfügen von Spillcode

Damit tatsächlich immer auf den korrekten Inhalt eines Datenobjekts zugegriffen wird, muss es an allen Kanten des Kontrollflussgraphen, an denen sich sein Speicherort ändert, in den jeweiligen Speicher kopiert werden. Dies wird durch Spillcode erledigt, der an den entsprechenden Stellen eingefügt werden muss. Der LLIR müssen also Instruktionen hinzugefügt werden. Um die korrekten Stellen für das Kopieren der Objekte ermitteln zu können, müssen zunächst die innerhalb des Kontrollflussgraphen auf Knotenebene gegebenen Informationen auf die Kanten der LLIR übertragen werden.

Dazu wird nacheinander jede Kante zwischen Basisblöcken der LLIR betrachtet. Da für jeden Block der Knoten bekannt ist, in dessen Untergraph er enthalten ist, kann für alle Kanten bestimmt werden, welche Datenobjekte dort in den bzw. aus dem Scratchpad-Speicher kopiert werden müssen. Damit kann für jede Kante der benötigte Spillcode erzeugt werden. Abbildung 6.4 verdeutlicht dieses Vorgehen und zeigt, warum diese Übertragung der Informationen auf die Kanten der LLIR notwendig ist.

Für die Schleife aus den Knoten b und c wurde die Auslagerung der Datenobjekte Y und Z bestimmt, im umschließenden Graphen soll sich X im Scratchpad-Speicher befinden. Da die Schleife jedoch zwei Austrittskanten besitzt, die in der verschachtelten Darstellung nicht mehr ersichtlich sind, muss jede Kante der LLIR betrachtet werden und für diese die benötigten Kopieroperationen bestimmt werden. Erst dann kann der Spillcode erzeugt und an den entsprechenden Kanten

eingefügt werden. In der Abbildung ist dieser durch nur einen Knoten dargestellt, tatsächlich können jedoch zahlreiche Basisblöcke nötig sein, um die Objekte zu kopieren.

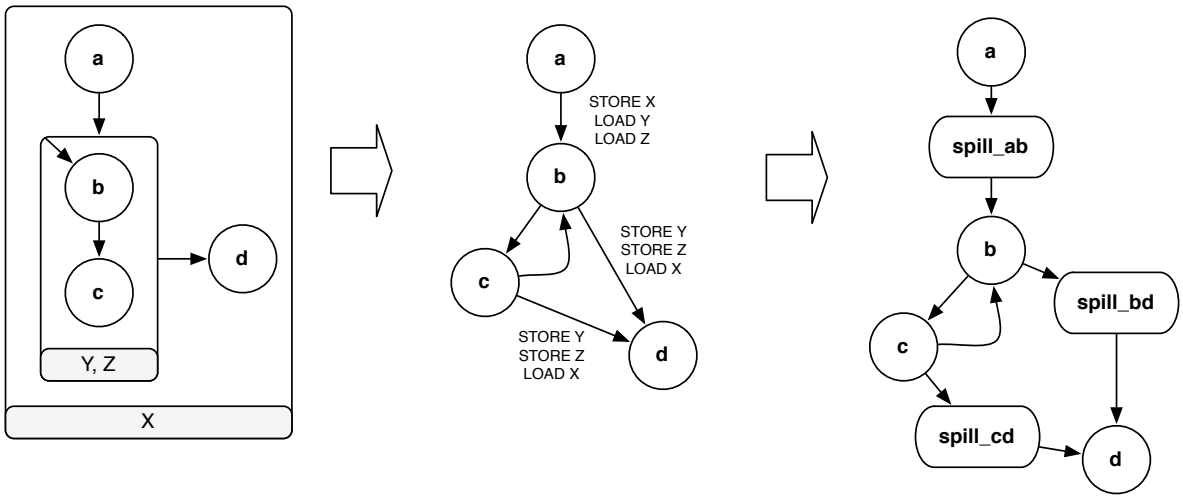

Abbildung 6.4: Einfügen von Spillcode an den Kanten der LLIR

Bei genauerer Betrachtung der Abbildung wird deutlich, dass in diesem Fall der Spillcode an den Kanten $c \rightarrow d$ und $b \rightarrow d$ exakt die gleiche Aufgabe erfüllt und somit eigentlich nur ein Mal erzeugt werden müsste. Da die aktuelle Implementierung jedoch die Kanten des Kontrollflussgraphen unabhängig voneinander betrachtet, kann dies nicht berücksichtigt werden. Daraus resultiert eine geringfügig größere Binärdatei, als eigentlich nötig wäre. Dieser Spezialfall tritt allerdings nur bei Schleifen mit mehreren Austrittspunkten auf.

Das Generieren und Einfügen von Spillcode ist ein sehr technischer Vorgang, der hier nur kurz umrissen werden soll. Der erzeugte Code kann zahlreiche Basisblöcke enthalten, die dann an geeigneter Stelle in die LLIR eingefügt werden müssen. Dazu ist zu prüfen, ob die entsprechende Kante einen Sprung oder Funktionsaufruf darstellt. In diesem Fall muss der Spillcode vor der Sprung- bzw. Call-Instruktion eingefügt werden, es ist also ein Auftrennen des zugehörigen Basisblocks erforderlich. Zusätzlich müssen die Inhalte von Registern, die innerhalb des Codes genutzt werden, auf dem Stack gesichert werden. Der erzeugte Spillcode hat somit folgenden Aufbau:

1. Sichern der im Spillcode benutzten Register auf dem Stack
2. Kopieren aller Datenobjekte in den SRAM, die im folgenden Block nicht mehr im Scratchpad gespeichert sind (falls nötig, weil diese verändert wurden)
3. Kopieren aller einzulagernden Datenobjekte in den Scratchpad-Speicher
4. Wiederherstellen der ursprünglichen Registerinhalte
5. Durchführung des Sprungs bzw. Funktionsaufrufs (falls nötig)

Das tatsächliche Kopieren der Datenobjekte sollte natürlich möglichst effizient geschehen. Daher wurde in der Implementierung die *Zero Overhead Loop* Funktionalität der TriCore Architektur

genutzt, mit deren Hilfe eine Schleife innerhalb einer eigenen Pipeline behandelt wird und die Prüfung der Abbruchbedingung keine weiteren Kosten verursacht. Die Speicherzugriffe nutzen zudem die *Postinkrement* Adressierungsart, daher müssen Quell- und Zieladresse nur einmal vor der Schleife explizit berechnet werden. Dies spart für jeden Schleifendurchlauf weitere Zyklen, die sonst zur Berechnung der jeweils nächsten Adressen nötig wären.

Innerhalb der Schleife werden jeweils 32 Bit Werte aus dem Quellspeicher gelesen und in den Zielspeicher geschrieben. Ist die Größe eines Datenobjekts in Byte nicht durch 4 teilbar, müssen im Anschluss daran also eventuell noch die restlichen Bytes kopiert werden. Dafür werden maximal 4 zusätzliche Operationen benötigt (durch das Laden und Speichern jeweils eines 16 Bit Wertes und eines 8 Bit Wertes).

Der so erzeugte Spillcode besitzt also eine sehr regelmäßige Struktur. Mit Hilfe der Informationen, die im Zuge der Untersuchung des Pipeline Verhaltens bei Load/Store Instruktionen (s. Kapitel 5.4.2) gewonnen wurden, können somit ausschließlich aus der Größe eines Datenobjekts die Kosten bestimmt werden, die ein Kopieren in den bzw. aus dem Scratchpad-Speicher verursacht. Dies wird genutzt, um innerhalb des ILP realistische Werte für die Variablen $loadcost_{n,d}$ und $storecost_{n,d}$ zu erhalten. Aufgrund von Nebeneffekten innerhalb der Pipeline können diese Werte zwar lediglich eine Abschätzung darstellen, durch Messungen mit Hilfe von aiT wurde jedoch festgestellt, dass nur eine geringe Abweichung zu den tatsächlichen Kosten auftritt.

Wurde an allen Kanten, an denen sich die Scratchpad-Belegung ändert, Spillcode eingefügt, sind alle Schritte vollzogen, um die dynamische Auslagerung von Datenobjekten innerhalb der LLIR umzusetzen. Bei der Ausführung wird das Programm nun für Schleifen und Funktionsaufrufe selbstständig den Scratchpad-Inhalt herstellen, der durch das ILP als besonders günstig gewählt wurde. Dennoch bleibt für jeden Basisblock die Kenntnis erhalten, auf welche Speicherbereiche er zugreift, eine WCET-Abschätzung ist also weiterhin mit der gleichen Genauigkeit möglich, wie für ein nicht modifiziertes Programm.

6.4 Einschränkungen

Die Erweiterung der ILP-Formulierung berechnet unter Berücksichtigung der Spillcode-Kosten eine dynamische Scratchpad-Allokation. Zusätzlich zu den Beschränkungen in der Abbildung des Kontrollflusses, die in Kapitel 4.6 beschrieben wurden, treten bei der dynamischen Optimierung weitere Probleme auf, die in diesem Abschnitt erläutert werden sollen.

Einige dieser Einschränkungen ergeben sich durch die beschriebene Art der ILP-Formulierung und lassen sich daher bei dem gewählten Ansatz nicht vermeiden. Andere ergeben sich lediglich aus der momentanen Form der Umsetzung und könnten durch zusätzlichen Aufwand bei der Implementierung, für den im Rahmen dieser Arbeit leider keine Zeit mehr zur Verfügung stand, behoben werden.

Beschränkung auf Schleifen und Funktionen

Eine Änderung des Scratchpad-Inhaltes ist nur an den Grenzen von Schleifen und Funktionen möglich, da nur diese verschachtelte Knoten im Graphen darstellen können. Das Ermöglichen von Kopiervorgängen an einzelnen Kanten könnte bessere Ergebnisse liefern, wenn innerhalb eines Programmabschnitts ohne Schleifen zahlreiche unterschiedliche Datenobjekte genutzt werden.

Dies ist jedoch nicht mit dem Konzept der verschachtelten Knoten vereinbar, das für die implizite Minimierung der WCET durch das ILP notwendig ist. Es ist allerdings davon auszugehen, dass sich die Menge der durch einen Programmabschnitt referenzierten Variablen gerade an Schleifen- und Funktionsgrenzen ändert.

Fragmentierung des Scratchpad-Speichers

Der Algorithmus berechnet für jeden verschachtelten Knoten die Menge der Datenobjekte, die für diesen in den Scratchpad-Speicher ausgelagert werden. Dessen Größenbeschränkung wird durch Kapazitäts-Constraints ausgedrückt, die an jeder Stelle sicherstellen, dass die Summe der Größen der einzelnen Datenobjekte nicht die Größe des Speichers überschreitet. Was hierbei leider nicht berücksichtigt werden kann, ist, dass einige Bereiche durch Fragmentierung nicht nutzbar sind und somit eventuell nicht alle Datenobjekte, für die eine Auslagerung bestimmt wurde, tatsächlich in den schnelleren Speicher kopiert werden können. An einem simplen Beispiel lässt sich dies leicht verdeutlichen (siehe Abbildung 6.5).

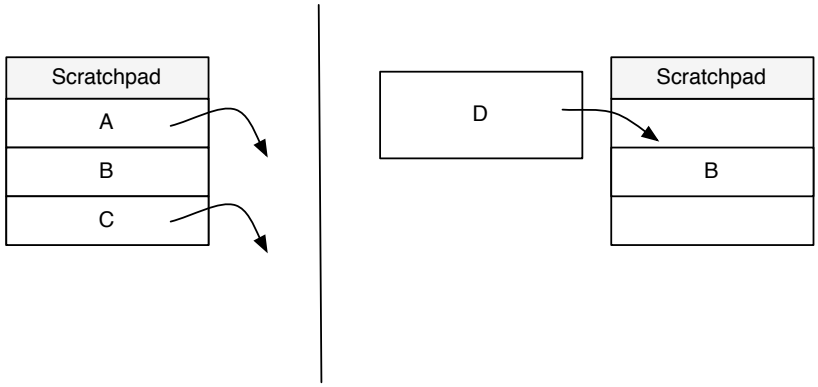

Abbildung 6.5: Fragmentierung des Scratchpad-Speichers

Hier werden die Datenobjekte A und C aus dem Scratchpad-Speicher kopiert. Damit wird theoretisch genügend Platz frei, um D einzulagern. Leider ist durch die ungünstige Platzierung von B jedoch kein durchgehend freier Speicherbereich verfügbar, der D fassen kann. Ein Verschieben von B würde die Einlagerung von D zwar ermöglichen, aber auch zusätzliche Kopierkosten verursachen.

Durch geschickte Wahl der Speicheradressen von Datenobjekten lässt sich dieses Problem zumindest abschwächen. Der in Abschnitt 6.3.2 beschriebene Algorithmus zur Berechnung von Scratchpad-Adressen erlaubt die Zuweisung unterschiedlicher Scratchpad-Adressen für ein Datenobjekt an unterschiedlichen Programmstellen. Dadurch treten weniger Konflikte um den freien Speicher auf als bei der Verwendung von festen Adressen.

Aufruf einer Funktion von unterschiedlichen Programmstellen aus

Die größte Einschränkung des gewählten Ansatzes ergibt sich beim Aufruf der selben Funktion von unterschiedlichen Programmstellen aus. Durch das ILP wird für jede Funktion genau eine günstige Scratchpad-Belegung gewählt. In diesem Zusammenhang ergeben sich 2 Probleme.

Das erste Problem betrifft die Übergabe von Zeigern. Wird einer Funktion ein Zeiger auf ein globales Datenobjekt übergeben, so kann dieser bei Aufrufen von unterschiedlichen Programmstellen aus auch auf verschiedene Datenobjekte zeigen. Somit können je nach Aufruf komplett unterschiedliche Scratchpad-Belegungen sinnvoll sein. Es kann jedoch nur eine Belegung für die Funktion gewählt werden. Da dennoch für jeden Aufruf der entsprechende Spillcode eingefügt werden muss, fallen dessen Kosten in jedem Aufruf an, der Gewinn jedoch nur in dem Aufruf, bei dem das ausgelagerte Datenobjekt tatsächlich referenziert wird.

Das zweite Problem ergibt sich sogar beim Aufruf von Funktionen, die gar keine Datenzugriffe durchführen. Für diese muss dennoch innerhalb des ILP eine Scratchpad-Belegung bestimmt werden. Wird eine solche Funktion nur an einer Stelle aufgerufen, so wird für diese die Belegung der aufrufenden Funktion übernommen, da in diesem Fall kein Spillcode erzeugt werden muss, also keine Kosten entstehen. Wird eine Funktion von mehreren Stellen aus aufgerufen, so ist dies nicht mehr in jedem Fall möglich. Wenn sich die günstigsten Belegungen der aufrufenden Funktionen unterscheiden, muss nun an mindestens einem der Funktionsaufrufe Spillcode eingefügt werden. Somit existiert eine Abhängigkeit zwischen allen aufrufenden Funktionen. Dieses Problem soll an einem Beispiel gezeigt werden:

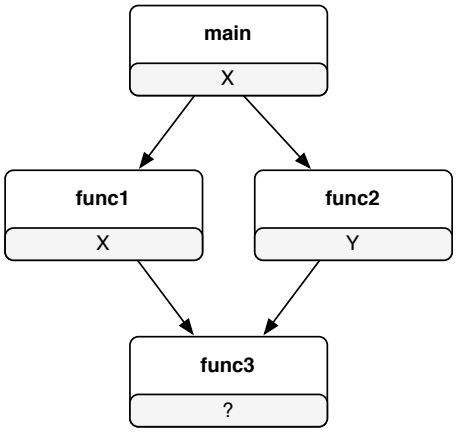

Abbildung 6.6: Aufruf einer Funktion von unterschiedlichen Programmstellen aus

In Abbildung 6.6 ist ein einfacher Call-Graph dargestellt. Es wird davon ausgegangen, dass innerhalb der Funktionen keine Schleifen existieren und somit nur für die 4 Funktionen Scratchpad-Belegungen berechnet werden. Weiterhin existieren die Datenobjekte X und Y, welche jeweils den gesamten Scratchpad-Platz benötigen. Die Funktion $func3$ wird an zwei Programmstellen aufgerufen. In $func1$ wäre eine Auslagerung von X günstig, in $func2$ eine Auslagerung von Y. Wird nun in $func3$ X im Scratchpad gespeichert, so müsste für den Aufruf $func2 \rightarrow func3$ Spillcode zum Auslagern von Y und Einlagern von X eingefügt werden. Dies ist auch dann nötig, wenn $func3$ auf keines der beiden Datenobjekte zugreift. Im Fall einer Auslagerung von Y in $func3$ wäre entsprechender Spillcode für den Aufruf $func1 \rightarrow func3$ nötig. Betrachtet man das gesamte Programm, so sind die Kosten des auf diese Weise entstehenden Spillcodes meist höher als der Gewinn, der durch eine dynamische Auslagerung erzielt werden kann.

Durch die gegebene Formulierung des ILP wird daher in diesen Fällen meist eine Lösung berechnet, die gar keine Auslagerung der betroffenen Datenobjekte beinhaltet, da nur so die Erzeugung von überflüssigem Spillcode vermieden werden kann. In der Praxis führt dies häufig dazu, dass

durch die dynamische Allokation exakt die selben Ergebnisse erzielt werden wie durch die statische Allokation, da nur für die *main*-Funktion eine Auslagerung bestimmt wird und diese für den gesamten Programmverlauf beibehalten wird.

Dieses Problem lässt sich durch das Kopieren aller Funktionen, die von unterschiedlichen Programmstellen aus aufgerufen werden, lösen. Für jeden Aufruf existiert dann eine eigene Kopie dieser Funktion, für die eine Scratchpad-Belegung gewählt wird. Natürlich hat dieser Ansatz deutliche Nachteile, insbesondere die daraus resultierende Vergrößerung des Codes ist nicht zu vernachlässigen. Zudem ist eine solche Anpassung im WCC bisher nicht implementiert, das zu übersetzende Programm muss also von Hand angepasst werden.

Diese Einschränkung behindert den praktischen Einsatz der dynamischen Scratchpad-Allokation in der beschriebenen Form. Durch die Investition weiterer Forschungsarbeit sollte geklärt werden, ob eine Verbesserung des Ansatzes möglich ist, welche die beschriebenen Probleme nicht aufweist. Zumindest aber könnte das Vervielfachen der Funktionen programmatisch erfolgen, so dass keine Anpassung des Quellcodes mehr nötig ist.

Adressberechnungen und Speicherzugriff in verschiedenen Knoten

Wurde für ein Datenobjekt die Auslagerung in den Scratchpad-Speicher bestimmt, so müssen alle Instruktionen, welche dieses referenzieren, angepasst werden, um auf den neuen Speicherort im Scratchpad zu verweisen. Leider ist dies keine triviale Aufgabe, denn die Berechnung der Adressen kann zu einem sehr viel früheren Zeitpunkt geschehen, als der tatsächliche Lade- oder Speichervorgang. Dafür gibt es zwei unterschiedliche Ursachen:

- Vielen Funktionen werden Zeiger auf bestimmte Datenobjekte übergeben. Diese zeigen dann konsequenterweise auf die Speicherstelle, an der das entsprechende Objekt in der aufrufenden Funktion liegt. Wurde für die aufgerufene Funktion eine Auslagerung des Objekts bestimmt, ist die übergebene Adresse somit nicht mehr korrekt. Da die Nutzung des entsprechenden Zeigers jedoch zusätzlich auch in der aufrufenden Funktion möglich ist, darf sein Ziel erst zum Zeitpunkt des Funktionsaufrufs geändert werden. Es ist allerdings im Allgemeinen auf LLIR-Ebene nicht möglich, den Inhalt eines per Adressregister oder Stack übergebenen Zeigers zu bestimmen. Daher stehen nicht genügend Informationen bereit, um eine solche Änderung durchzuführen.
- Speicherzugriffe, deren Ziel fest definiert ist, werden vom Code Selector durch zwei aufeinanderfolgende Instruktionen realisiert: das Laden eines Teils der Adresse in ein Adressregister und den tatsächlichen Zugriff auf den in diesem Register gespeicherten Wert (eventuell zuzüglich eines Offsets). Durch die zahlreichen Optimierungen auf LLIR-Ebene können diese Instruktionen jedoch verschoben werden. Gerade beim Zugriff auf ein Array innerhalb einer Schleife kann es sehr vorteilhaft sein, die Basisadresse vor dem Betreten der Schleife in ein Adressregister zu laden, da so pro Schleifendurchlauf eine Instruktion gespart werden kann. Dies wird durch die WCC-Optimierung *loop invariant code motion* durchgeführt. Wurde jedoch eine Auslagerung nur für den Schleifenkörper bestimmt, so würde das Adressregister in einem solchen Fall wieder auf die falsche Speicherstelle zeigen. Bei verschachtelten Schleifen kann die Initialisierung des Registers zahlreiche Ebenen oberhalb des tatsächlichen Zugriffs erfolgen. Es ergibt sich somit das gleiche Problem wie im ersten Punkt.

Eine Lösung für diese Probleme konnte innerhalb der für diese Arbeit zur Verfügung stehenden Zeit leider nicht gefunden werden. Um dennoch eine WCET-Abschätzung des erzeugten Codes zu ermöglichen, wird daher mittels Annotationen das Zugriffsziel aller Load/Store Instruktionen, die auf ein in den Scratchpad-Speicher ausgelagertes Objekt zugreifen, an aiT übergeben. Die WCET-Analyse berücksichtig auf diese Weise die korrekten Zugriffsziele, für die resultierende Binärdatei kann allerdings die Korrektheit nicht garantiert werden.

Kapitel 7

Resultate

In diesem Kapitel sollen die Resultate aufgezeigt werden, dich sich mit den vorgestellten Optimierungsalgorithmen erzielen lassen. Da der Erfolg einer Scratchpad-Allokation von Daten sehr stark von dem zu übersetzenden Programm abhängt, wurden eine ganze Reihe unterschiedlicher Benchmarks untersucht, die verschiedene in eingebetteten Systemen auftretende Aufgabenbereiche abdecken. Diese werden im ersten Teil dieses Kapitels beschrieben.

Die statische Allokation ist in der in dieser Arbeit vorgestellten Form sehr ausgereift und lässt sich auf nahezu beliebige Programme anwenden. Daher konnten alle vorgestellten Benchmarks auf den so zu erzielenden Gewinn überprüft werden. Diese Resultate werden im zweiten Abschnitt beschrieben.

Für die dynamische Allokation, die ein optionales Ziel dieser Arbeit darstellt, gilt dies leider nicht. Eine Anwendung ist hier zwar auch auf alle Benchmarks möglich, ohne Modifikation des Quellcodes werden jedoch aufgrund der in Kapitel 6.4 beschriebenen Einschränkungen keine anderen Ergebnisse erzielt als durch die statische Allokation. Auch nach Anpassung des Quellcodes konnten allerdings keine aussagekräftigen Ergebnisse erzeugt werden, da durch das Einfügen von Spillcode Nebeneffekte eintreten, die für die Berechnung von nicht vergleichbaren Werten durch aiT verantwortlich sind. Im dritten Abschnitt dieses Kapitels wird daher beschrieben, welche Probleme sich bei der Messung der durch die dynamische Allokation erzielten WCET-Werte ergaben.

Es ist zu beachten, dass die in diesem Kapitel genannten Zahlen nicht die tatsächliche WCET darstellen, sondern die durch aiT ermittelte WCET-Abschätzung. Es ist also durchaus möglich, dass in einigen Fällen tatsächlich bessere Ergebnisse erzielt werden, als durch diese Zahlen belegt wird, da aiT immer eine sichere Abschätzung ausgibt.

7.1 Benchmarks

Um den Gewinn einer Optimierung zu testen, werden Benchmarks benötigt, die möglichst praxisnahe Anwendungsfälle darstellen, um realistische und allgemeingültige Ergebnisse zu erzielen. Für den WCC existiert eine Benchmark-Suite, die eine ganze Reihe von Programmen enthält und speziell auf die Untersuchung von WCET-Optimierungen ausgelegt wurde. Insbesondere sind Informationen wie Schleifeniterationsgrenzen und Rekursionstiefen bereits im Quelltext

annotiert, so dass die WCET-Analyse durch aiT direkt aus dem WCC heraus durchgeführt werden kann. Die WCC Benchmark-Suite basiert u.a. auf MediaBench [LPMS97], NetBench [MMSH01], UTDSP [Lee08] und der MRTC Real-Time Benchmark-Suite [mrt08].

Aus dieser großen Auswahl an Benchmarks wurden für die Untersuchung der in dieser Arbeit vorgestellten Optimierung nur solche gewählt, die mindestens eine globale bzw. statische Variable besitzen, da ansonsten kein Gewinn durch die Scratchpad-Allokation erzielt werden kann. Leider terminiert die in ICD-C enthaltene Aliasanalyse nicht für alle Benchmarks. Das Problem konnte in der zur Verfügung stehenden Zeit nicht gelöst werden, so dass auch die entsprechenden Programme nicht untersucht werden konnten, da für diese keine ausreichenden Informationen über die Datenzugriffe der einzelnen Instruktionen zur Verfügung standen.

Die Tabelle 7.1 enthält eine Übersicht aller für diese Arbeit untersuchten Benchmarks. Für jedes Programm ist dort dessen Einsatzzweck, die Zahl der Codezeilen (*LoC, Lines of Code*), die Anzahl der globalen bzw. statischen Variablen und die Größe alle globalen Daten in Bytes aufgeführt. Da viele Benchmarks ursprünglich zu anderen Zwecken genutzt wurden und nachträglich zur Bestimmung der WCET angepasst wurden, finden sich in deren Quelltext teilweise Variablen, die nicht mehr referenziert werden. Diese wurden für die Angaben in der Tabelle nicht mitgezählt, da sie schon in einer frühen Optimierungsstufe des Compilers entfernt werden und somit gar nicht mehr in der Eingabe des Algorithmus auftreten.

Anhand der Tabelle wird deutlich, dass ein sehr breites Spektrum von Benchmarks untersucht werden konnte. Einige Programme besitzen viele kleine Datenobjekte, während andere nur ein oder zwei große Arrays global speichern. Dies resultiert aus der unterschiedlichen Verwendung globaler Variablen in den einzelnen Programmen: Häufig werden lediglich Ein- und Ausgabe global gespeichert, während alle übrigen Variablen lokal deklariert werden. Andere Benchmarks speichern auch in Zwischenberechnungen verwendete Werte im globalen Speicher. Auch der Codeumfang unterscheidet sich zwischen den einzelnen Benchmarks deutlich, lässt aber keine Rückschlüsse auf die resultierende WCET zu. Einige Programme bestehen lediglich aus einer Schleife, die jedoch über ein sehr großes Array iteriert und so zahlreiche Zyklen benötigt. Andere bestehen aus vielen Funktionen, behandeln aber nur vergleichsweise wenig Eingabedaten.

Innerhalb des Scratchpad-Speichers des TC1796 stehen im WCC maximal 40960 Bytes für die Auslagerung von globalen Daten zur Verfügung. Der übrige Platz wird verwendet, um Stack und CSA zu speichern. Da keiner der verwendeten Benchmarks überhaupt eine solche Menge an globalen Daten nutzt, kann die Qualität der Algorithmen zur Scratchpad-Allokation nicht beurteilt werden, ohne den zur Verfügung stehenden Speicher künstlich zu begrenzen. Alle Benchmarks wurden daher für unterschiedliche Scratchpad-Größen zwischen 8 Bytes und 40960 Bytes untersucht. Für den höchsten Wert lässt sich so immer erkennen, welche WCET-Einsparung maximal möglich ist, wenn alle Datenobjekte des Programms in den Scratchpad-Speicher ausgelagert werden.

Die Vielfalt der untersuchten Benchmarks ermöglicht die Beurteilung der durch die Scratchpad-Allokation erzielten WCET-Verbesserung für unterschiedliche Anwendungsfälle, so dass realistische Aussagen über den im Durchschnitt zu erwartenden Gewinn der Optimierungen getroffen werden können.

Name	Beschreibung	LoC	# glob. Daten	Größe glob. Daten
adpcm_decoder	Audio Codec	768	74	1288
adpcm_encoder	Audio Codec	824	72	1364
binarysearch	Binäre Suche	117	2	124
bsort100	Sortieralgorithmus	119	2	408
compressdata	Kompressionsalgorithmus	381	9	133
countnegative	Zählen negativer Werte in einer Matrix	135	6	1620
crc	Prüfsummenberechnung	158	5	1042
edge_detect	Kantendetektion	415	10	33607
fdct	Diskrete Cosinustransformation	238	1	128
fft1	Fast Fourier Transformation	246	2	64
fir	Finite Impulse Response Filter	329	2	2948
g721_encode	Audio Codec	1520	10	2192
g723_encode	Audio Codec	1534	10	2288
gsm_encode	Audio Codec	3178	10	8110
h263	Video Codec	179	12	39440
hamming_window	Digitale Signalanalyse	100	3	1844
insertsort	Sortieralgorithmus	143	1	44
jfdctint	Diskrete Cosinustransformation	441	1	256
lcdnum	Ausgabe v. Zahlen auf einem LCD Display	118	2	2
lms	Signalverbesserung in Audiodaten	290	10	1120
ludcmp	Lösen einer linearen Gleichung	204	3	1400
matmult	Matrixmultiplikation	152	4	4804
md5	Prüfsummenberechnung	598	2	65
ndes	Verschlüsselungsalgorithmus	497	13	1262
petrinet	Simulation eines Petri-Netzes	901	6	72
qurt	Lösen einer quadratischen Gleichung	186	4	32
searchmultiarray	Suche in einem mehrdimensionalem Array	605	2	5000
selection_sort	Sortieralgorithmus	100	1	1200
st	Statistische Berechnungen auf 2 Arrays	184	6	8016
statemate	Steuerung eines elektrischen Fensterhebers	1223	98	227
test3	Durchlaufen von 8 großen Arrays	4126	9	32772

Tabelle 7.1: Beschreibung der Benchmarks

7.2 Resultate der statischen Optimierung

Zur Untersuchung der WCET-Verbesserung, die sich durch die statische Scratchpad-Allokation von Daten erzielen lässt, wurde die Optimierung auf allen im vorherigen Abschnitt aufgeführten Benchmarks ausgeführt. Dabei ergeben sich sehr unterschiedliche Resultate. Während einige Programme kaum von der Optimierung profitieren, können bei anderen beträchtliche Verbesserungen erzielt werden. Von Bedeutung ist jedoch nicht nur der bei maximaler Scratchpad-Größe zu erzielende Gewinn, sondern insbesondere die Veränderung der Werte bei steigender Scratchpad-Größe. Anhand dieser lässt sich erkennen, dass der Algorithmus tatsächlich genau die Datenobjekte einlagert, mit deren Hilfe sich für die jeweilige Größe der maximale Gewinn erzielen lässt. Es konnte kein Fall beobachtet werden, in dem bei steigender Scratchpad-Größe eine Verschlechterung der WCET auftritt.

Die enge Integration von Optimierung und WCET-Analyse in den WCC machten es möglich,

sämtliche Werte ausschließlich durch Aufruf der Compilers mit den entsprechenden Kommandozeilenflags zu ermitteln. Dazu wurde die Optimierungstufe $O1$ gewählt. Diese bietet einen guten Kompromiss zwischen optimiertem Code und schneller Übersetzungszeit. Zudem tritt bei Nutzung der höheren Optimierungsstufe $O2$ das Problem auf, dass die Aliasanalyse von ICD-C für einige weitere Benchmarks nicht terminiert.

Während die WCET-Analyse durch aiT für einige Benchmarks bis zu einer Stunde benötigt, beträgt die Laufzeit des Optimierungalgorithmus in keinem Fall mehr als einige Sekunden. Dies umschließt insbesondere auch die Zeit zur Lösung des ILP durch den LP-Solver `cplex`, obwohl teilweise weit über 2000 Variablen definiert werden. Der Algorithmus kann also durchaus als effizient bezeichnet werden.

7.2.1 Benchmark-Ergebnisse

An dieser Stelle sollen nun exemplarisch die Resultate für einige der Benchmarks detaillierter beschrieben werden. Da der WCET-Gewinn in Zyklen für einen Vergleich wenig aussagekräftig ist, wird in den abgebildeten Diagrammen von diesem abstrahiert und lediglich das jeweilige Verhältnis zur WCET ohne Auslagerung von Datenobjekten betrachtet. Für eine Auflistung der durch aiT ermittelten Werte sei auf Anhang A verwiesen.

Zunächst soll ein sehr typischer Fall betrachtet werden: Bei Benchmarks mit nur einem großen Datenobjekt ist ein sprunghafter Abfall der gemessenen WCET zu erkennen, sobald dieses Datenobjekt in den Scratchpad-Speicher passt. Zu erkennen ist dies am besten anhand des Benchmarks `bsort100`, die den Bubblesort-Algorithmus auf einem 404 Bytes großen Array durchführt. In Abbildung 7.1 ist zu erkennen, dass der Gewinn fast 30 Prozent beträgt, wenn das Array im schnelleren Speicher liegt. In dem Programm wird noch eine weitere 4 Bytes große Variable genutzt. Auf diese wird jedoch nur schreibend zugegriffen. Wie in Kapitel 5.4.2 beschrieben, können Schreibzugriffe auf den SRAM an bestimmten Stellen ohne ein Anhalten der Pipeline durchgeführt werden. Da für diese Variable ein solcher Fall vorliegt, kann durch ihre Auslagerung kein Gewinn erzielt werden.

Ein ähnliches Verhalten zeigen auch die übrigen Benchmarks mit nur einem großen globalen Datenobjekt, auch wenn bei diesen geringere Gewinne erzielt werden. Dazu gehören beispielsweise `selection_sort` (8,5% max. Gewinn) und `jfdctint` (8,2% max. Gewinn).

Bei dem Programm `petrinet` handlet es sich um automatisch aus einem Petri-Netz generierten C-Code. Es besitzt 6 globale Variablen, die zusammen nur 72 Bytes einnehmen. Auf diese wird allerdings bei der Ausführung häufig zugegriffen, so dass durch die Optimierung eine Verbesserung der WCET um bis zu 31,1% erzielt wird (s. Abb. 7.2).

Auch `insertsort`, die Implementierung eines weiteren Sortieralgorithmus, besitzt nur sehr wenige globale Daten (44 Bytes), dennoch kann eine Verbesserung der WCET um bis zu 21,2% erreicht werden. Ähnliches gilt für `md5` (65 Bytes, 23,1% max. Gewinn). Bei anderen Benchmarks mit wenigen globalen Daten hingegen werden auch nur sehr geringe Gewinne erzielt, wie z.B. für `fft1` (64 Bytes, 1,37% max. Gewinn) oder `qurt` (32 Bytes, 0,84% max. Gewinn). Aus dem Umfang der globalen Datenobjekte lassen sich also keine Rückschlüsse darauf ziehen, welcher Erfolg sich durch die Scratchpad-Allokation erreichen lässt. Dies ist hauptsächlich abhängig von der Anzahl der Zugriffe auf diese Daten.

Bei Benchmarks mit sehr vielen Datenobjekten kann eine stetige Verbesserung der WCET bei

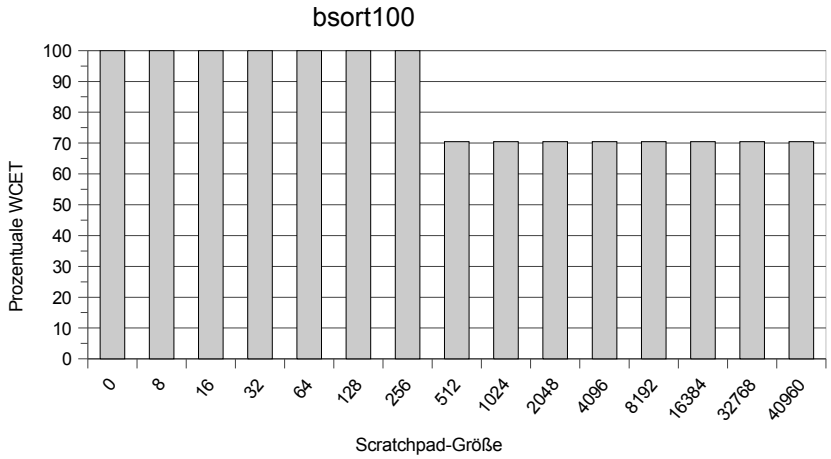

Abbildung 7.1: Resultate für `bsort100`

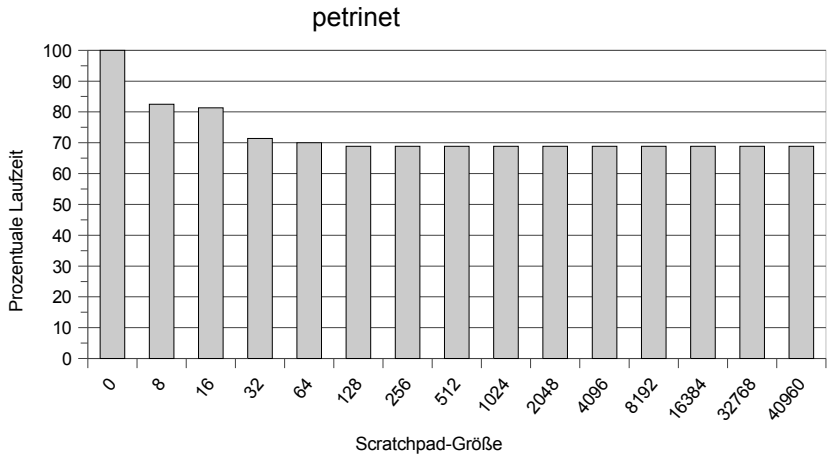

Abbildung 7.2: Resultate für `petrinet`

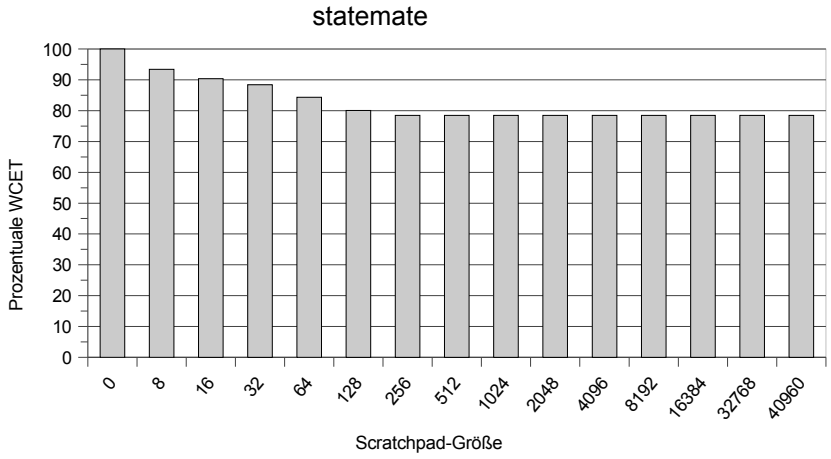

Abbildung 7.3: Resultate für `statemate`

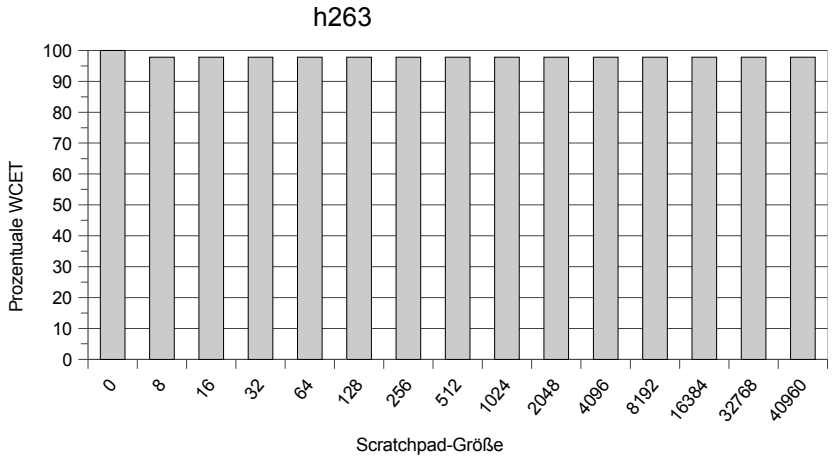

Abbildung 7.4: Resultate für `h263`

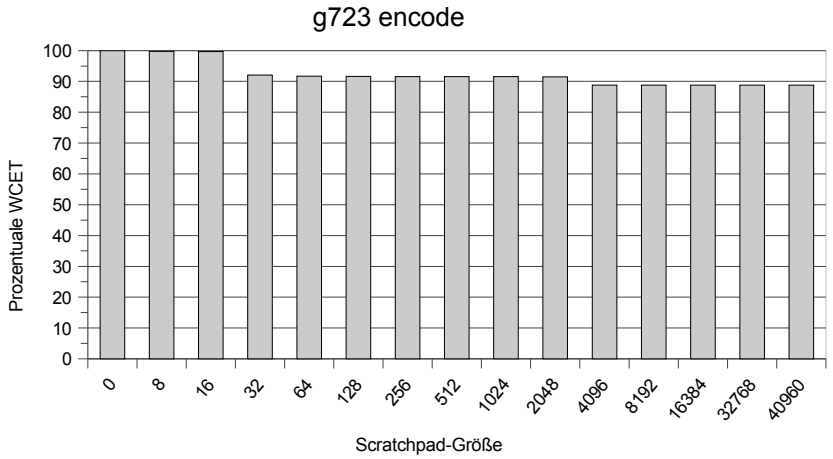

Abbildung 7.5: Resultate für `g723_encode`

Abbildung 7.6: Resultate für `test3`

steigender Scratchpad-Größe beobachtet werden. Ein gutes Beispiel hierfür ist `statemate` (s. Abb. 7.3). Der Code dieses Programms wurde automatisch aus einem Zustandsübergangsdiagramm generiert und beschreibt die Steuerung eines elektrischen Fensterhebers. 98 globale Variablen, davon keine größer als 4 Bytes, speichern den aktuellen Zustand des Diagramms. Auf einige wird dabei häufig zugegriffen, auf andere nur sehr selten. So ist zu erklären, dass schon bei einer Scratchpad-Größe von nur 8 Bytes eine Verbesserung der WCET um 6,5% gemessen werden kann. Die maximale Verbesserung liegt hier ab einer Scratchpad-Größe von 256 Bytes bei 21,4%. Obwohl der zur Verfügung stehende Scratchpad-Speicher in diesem Fall 32 mal größer ist als 8 Bytes, ist der Gewinn nur um den Faktor 3,3 höher. Dies belegt, dass durch den Algorithmus tatsächlich jene Datenobjekte gewählt werden, durch deren Auslagerung sich der größte Gewinn erzielen lässt.

Aufgabe des Benchmarks `h263` (s. Abb. 7.4) ist die sehr rechenintensive Umrechnung zwischen verschiedenen Farbräumen bei der Dekodierung von Videodaten. Obwohl hier eine sehr große Anzahl globaler Daten (39440 Bytes) referenziert wird, liegt die maximale Verbesserung bei nur 2,18%. Dies liegt allerdings daran, dass aufgrund von komplexen Zeigeroperationen bei der Übergabe von Funktionsparametern nicht alle Datenzugriffe erkannt werden können. Weder die ICD-C Aliasanalyse, noch die durch aiT durchgeführte Wertanalyse können das Ziel der wesentlichen Speicherzugriffe ermitteln, so dass diese weder innerhalb des ILP noch bei der WCET-Analyse bekannt sind. Auch bei Auslagerung aller Datenobjekte muss aiT aufgrund der nicht bekannten Zugriffsziele vom ungünstigsten Fall, also einem Zugriff auf den langsamsten Speicher ausgehen. Daher fällt der Gewinn entsprechend gering aus. Durch Anpassung des Quellcodes ließe sich dieses Problem umgehen, es soll jedoch an dieser Stelle die Anwendbarkeit des Algorithmus auf beliebige Programme untersucht werden.

Eine ähnlich aufwändige Berechnung ist die Kodierung von Sprachdaten. Diese ist Aufgabe des Benchmarks `g723_encode` (s. Abb. 7.5). Hier können allerdings die Datenzugriffe erkannt und daher bessere Ergebnisse erzielt werden. Der maximale Gewinn beträgt 11,2%.

Der Benchmark `test3` ist kein besonders realitätsnahes Programm, sondern es wurde zum Test der statischen WCET Analyse durch aiT entwickelt. Hier werden 8 Arrays in tief verschachtelten Funktionen durchlaufen. Anhand dieses Benchmarks lässt sich gut verdeutlichen, welche Ergebnisse bei wenig rechenintensiven Programmen, die fast ausschließlich aus Datenzugriffen bestehen, erzielt werden können. In Abbildung 7.6 ist ersichtlich, dass bei Auslagerung aller globalen Datenobjekte eine Verbesserung der WCET um 38% erreicht wird.

7.2.2 Fazit

Der Erfolg der statischen Scratchpad-Allokation von Daten hängt sehr stark von dem untersuchten Programm ab. Während für einige Benchmarks kaum Verbesserungen festgestellt werden konnten, wurden bei anderen Verbesserungen um über 30% gemessen. Dabei konnte kein Zusammenhang zwischen dem erzielten Gewinn und der Anzahl bzw. Größe der Datenobjekte oder dem Codeumfang festgestellt werden. Bei welcher Scratchpad-Größe die Optimierung am erfolgreichsten ist, kann daher auch nur bei Betrachtung eines bestimmten Programms ermittelt werden. Zu beobachten ist allerdings, dass auch bei kleinen Scratchpad-Größen häufig schon verhältnismäßig große Gewinne erzielt werden können.

Erwähnenswert ist, dass gerade bei den Benchmarks `statemate` und `petrinet`, deren Code durch visuelle Modellierungstools automatisch erzeugt wurde, sehr gute Ergebnisse erzielt wer-

den können. Da in einem solchen Fall der Entwickler selbst typischerweise keine Codeoptimierungen vornimmt (dies ist aufgrund der Struktur von generiertem Code häufig sehr mühsam), ist besonders hier eine vom Compiler durchgeführte Scratchpad-Optimierung sehr sinnvoll.

Leider können, wie das Beispiel `h263` zeigt, nicht in jedem Fall alle Datenzugriffsziele erkannt werden. Dies ist insbesondere bei der Verwendung von Zeigerarithmetik der Fall, da hier weder die Aliasanalyse von ICD-C noch die Wertanalyse von aiT die möglichen Variablen- bzw. Registerwerte korrekt verfolgen kann. Verfahren, die Datenzugriffsziele durch ein Profiling des zu optimierenden Programms berechnen, haben hier einen Vorteil. Allerdings können diese wiederum nicht auf einfache Weise innerhalb eines Compilers ohne Nutzerinteraktion integriert werden. Zudem erkennen diese Zugriffe nur innerhalb desjenigen Programmpfads, der während des Profiling-Durchlaufs genommen wird. Zugriffe innerhalb anderer Pfade, unter denen sich auch der Worst-Case Execution Pfad befinden kann, werden so nicht erkannt. Beim Schreiben eines Programms, das möglichst viel von der Scratchpad-Allokation profitieren soll, sollte also wenn möglich auf Zeigerarithmetik verzichtet werden.

Der durchschnittlich über alle 31 untersuchten Benchmarks erzielte Gewinn ist in Abbildung 7.7 dargestellt. Es ist zu sehen, dass dieser im Diagramm relativ konstant mit der Größe des Scratchpad-Speichers wächst. Dies liegt daran, dass für die x-Achse eine exponentielle Skalierung gewählt wurde. Tatsächlich ist das Verhältnis von erzieltem Gewinn zur Scratchpad-Größe bei den kleineren Werten deutlich größer. Es können also im Schnitt auch schon für sehr kleine Scratchpad-Größen gute Ergebnisse erzielt werden. Schon bei einer Größe von nur 8 Bytes kann ein durchschnittlicher Gewinn von 1,6% erreicht werden. Die Hälfte des maximal erreichbaren Gewinns, der im Schnitt bei 11,6% liegt, wird schon bei einer Größe von 256 Bytes mit 6,2% erstmals überschritten.

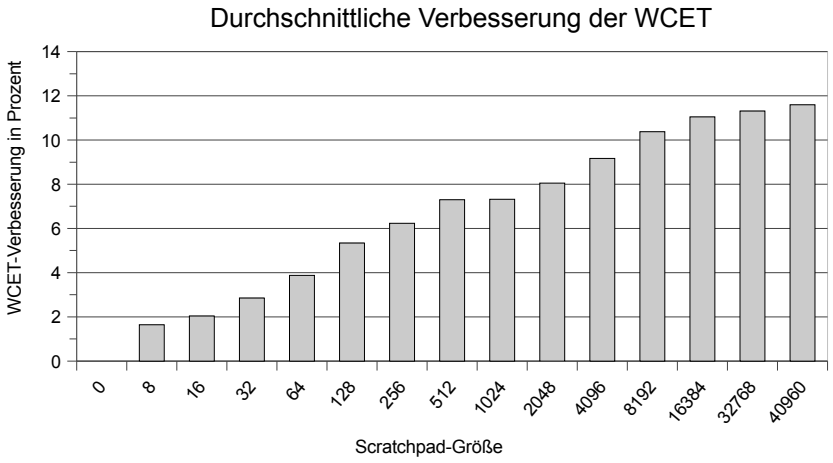

Abbildung 7.7: Durchschnittliche Verbesserung der WCET

Auf Grundlage dieser Daten kann die Aussage getroffen werden, dass die statische Scratchpad-Allokation von Daten zur WCET-Minimierung eine sehr lohnenswerte Optimierung ist. Dem Programmierer kann so viel Arbeit abgenommen werden, da die manuelle Untersuchung der Ausgabe von aiT und die anschließende Auswahl von auszulagernden Variablen keine triviale Aufgabe ist. Durch die nahtlose Einbindung in den Compiler kann die WCET eines Programms so ohne Mehraufwand beträchtlich verringert werden.

Da der WCC, in den die Optimierung integriert wurde, nur Code für den Infineon TriCore Prozessor erzeugt, konnten auch nur für diese Plattform Ergebnisse generiert werden. Interessant wäre in diesem Zusammenhang die Untersuchung, welche Gewinne sich auf anderen, weniger komplexen Prozessoren durch den entwickelten Algorithmus erzielen lassen. Insbesondere der relativ schnelle Zugriff auf den SRAM und die Möglichkeit von im Hintergrund laufenden Speichervorgängen sogen dafür, dass der TriCore Prozessor auch auf nicht im Scratchpad liegende Daten recht schnell zugreifen kann. In Systemen mit einem langsamen DRAM und blockenden Speicherzugriffen könnten vermutlich für die gleichen Benchmarks noch sehr viel bessere Ergebnisse erzielt werden.

7.3 Resultate der dynamischen Optimierung

Die im letzten Abschnitt vorgestellten Ergebnisse der statischen Scratchpad-Allokation belegen deren praktische Anwendbarkeit. Durch die als optionales Ziel dieser Arbeit definierte dynamische Allokation konnten leider keine vergleichbaren Ergebnisse erzielt werden. In diesem Abschnitt soll beschrieben werden, welche Beobachtungen bei der Untersuchung des Verhaltens der dynamischen Allokation gemacht wurden, und welche Probleme eine Berechnung von vergleichbaren Resultaten verhindern.

7.3.1 Eignung der Benchmarks für eine dynamische Optimierung

Ein Großteil der untersuchten Benchmarks ist für eine dynamische Optimierung nicht geeignet. Insbesondere Programme, die globale Datenobjekte nur zur Speicherung von Ein- und Ausgabe verwenden, können davon nicht profitieren, da hier meist innerhalb der selben Schleifen auf beide Arrays zugegriffen wird und somit eine dynamische Änderung des Scratchpad-Inhalts keinen Sinn ergeben würde. Aber auch für viele andere Benchmarks wird durch den Algorithmus keine dynamische Auslagerung vorgenommen. Die Resultate entsprechen in diesen Fällen exakt denen der statischen Allokation. Dies hat mehrere Gründe, die im Folgenden beschrieben werden sollen.

Hohe Spillcode-Kosten im Vergleich zum erzielbaren Gewinn

Auch Datenobjekte, deren Auslagerung im ersten Moment lohnenswert erscheint, eignen sich bei genauerer Betrachtung nicht immer dafür. Damit ein Datenobjekt in den Scratchpad-Speicher kopiert werden kann, muss zunächst der benötigte Platz geschaffen werden. Im ungünstigsten Fall müssen also bei Einlagerung eines n Bytes großen Datenobjekts zunächst $\frac{n}{4}$ 32-Bit Worte aus dem Scratchpad-Speicher zurück in den SRAM kopiert werden. Schon hierfür werden zwei Speicherzugriffe pro Wort benötigt (jeweils ein lesender und ein schreibender Zugriff). Erst dann kann das entsprechende Objekt in den Scratchpad-Speicher kopiert werden, was wiederum zwei Speicherzugriffe pro Wort verursacht. Somit sind in diesem Fall für den Spillcode insgesamt $\frac{n}{4} * 4$ Speicherzugriffe nötig.

Die TriCore Architektur erlaubt einen vergleichsweise schnellen Zugriff auch auf den SRAM. Insbesondere einzelne Schreibzugriffe auf diesen benötigen unter bestimmten Umständen genau wie Zugriffe auf den Scratchpad-Speicher nur einen Zyklus. Dies ist genau dann der Fall, wenn sie vereinzelt auftreten, also die vorausgehenden Operationen keinen Speicherzugriff vornehmen.

Im Programmcode kommt dies relativ oft vor, da hier häufig durch arithmetische Operationen ein Wert berechnet wird, der dann in den Speicher geschrieben wird. Innerhalb des Spillcodes jedoch werden zahlreiche Load/Store Instruktionen hintereinander ausgeführt, so dass die Schreibzugriffe nicht im Hintergrund durchgeführt werden können.

Aus diesen Gründen muss, damit durch die Auslagerung eines Objekts ein Gewinn erzielt werden kann, der höher ist als die Kosten des Spillcodes, sehr häufig auf das entsprechende Datenobjekt zugegriffen werden, während es im Scratchpad-Speicher liegt. Dies tritt leider nur in wenigen Fällen auf, da häufig benötigte Daten durch den WCC bevorzugt in Registern gehalten werden und der entsprechende Wert nur ein Mal geladen (und, falls er geändert wird, wieder zurückgeschrieben) werden muss.

Die Anzahl der Benchmarks, in denen eine dynamische Allokation überhaupt lohnenswert ist, ist also relativ gering.

Starke Abhängigkeit von der Scratchpad-Größe

Hinzu kommt, dass eine dynamische Allokation meist nur für ganz bestimmte Scratchpad-Größen lohnenswert ist. Sobald alle Datenobjekte in den schnelleren Speicher passen, besteht keine Veranlassung mehr, eine dynamische Auslagerung vorzunehmen. In diesem Fall entspricht das Ergebnis der dynamischen Allokation exakt dem der statischen Allokation. Für jeden Benchmark muss also genau das Intervall an Scratchpad-Größen untersucht werden, innerhalb dessen sich eine Auslagerung überhaupt lohnt. Eine Untersuchung aller Größen, die für die statische Allokation betrachtet wurden, ist hier nicht sinnvoll.

Benötigte Anpassungen aufgrund von mehrmals aufgerufenen Funktionen

In Kapitel 6.4 wurde erläutert, warum eine Anpassung von Programmen, in denen eine Funktion von unterschiedlichen Stellen aus aufgerufen wird, erforderlich ist. Eine Anpassung aller zuvor vorgestellten Benchmarks war leider aus Zeitgründen nicht mehr möglich. Im folgenden Abschnitt soll exemplarisch anhand eines einzelnen Benchmarks erläutert werden, welche weiteren Probleme auch nach Anpassung des Quellcodes bei der Ermittlung vergleichbarer WCET-Werte auftraten.

7.3.2 Ergebnisse des `ndes` Benchmarks

Das Verhalten der dynamischen Optimierung und die Probleme, die sich bei der Untersuchung der durch diese erzielten Ergebnisse ergeben, soll nun anhand des Benchmarks `ndes` beschrieben werden. Dieses Programm besitzt zahlreiche Schleifen mit vielen Iterationen, in denen auf jeweils unterschiedliche Datenobjekte zugegriffen wird. Somit erscheint eine Auslagerung bestimmter Daten in genau diesen Schleifen bei beschränktem Scratchpad-Speicher als lohnenswert. Dies kann auch in den Ausgaben des Algorithmus nachvollzogen werden: Bei bestimmten Scratchpad-Größen werden durch diesen tatsächlich entsprechende dynamische Änderungen bestimmt. Bei der Messung der so erzielten WCET-Einsparung treten allerdings Nebeneffekte auf, die eine vergleichende Bewertung der Resultate verhindern.

Der `ndes` Benchmarks greift auf 13 Datenobjekte der Gesamtgröße von 1262 Bytes zu. Daher

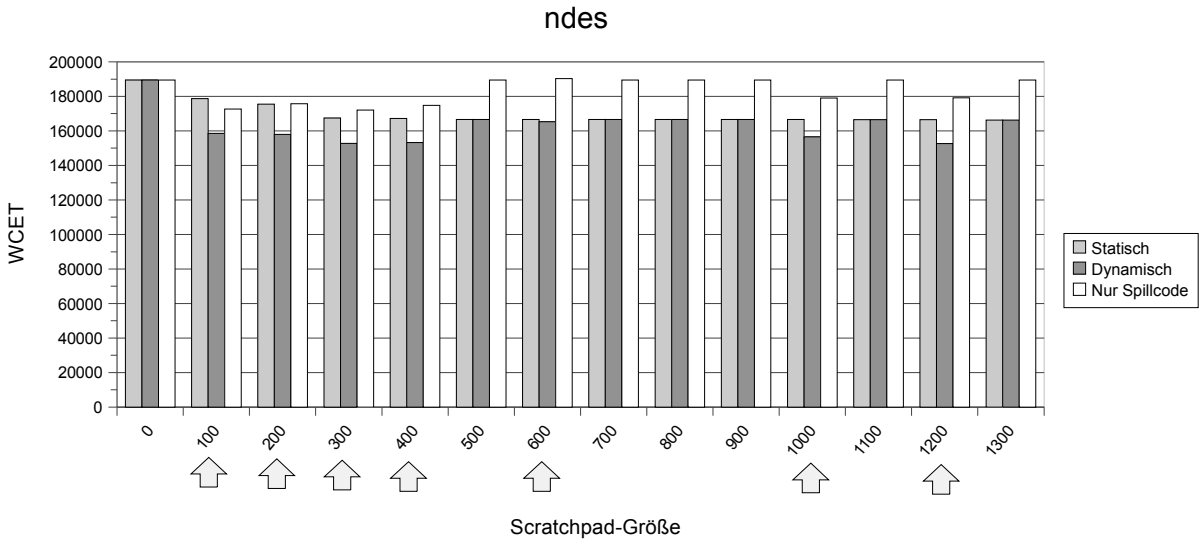

Abbildung 7.8: Untersuchung der dynamischen Allokation am Beispiel `ndes`

wurden Scratchpad-Größen von 0 bis 1300 Bytes in Abständen von je 100 Bytes untersucht. Zum Vergleich wurden auch die durch eine statische Allokation erzielten WCET-Werte für diese Größen gemessen. Während letztere wie zu erwarten mit steigender Scratchpad-Größe kontinuierlich abnehmen, weisen die gemessenen WCET-Werte der dynamischen Optimierung starke Schwankungen auf. Diese sind jedoch nicht auf ein Fehlverhalten des Algorithmus zurückzuführen, sondern auf Timing-Anomalien, die durch das Einfügen des Spillcodes verursacht werden.

Durch das Einfügen von Spillcode wird der Programmcode geändert, was für eine statische Allokation nicht nötig ist. Diese Änderung hat zur Folge, dass aiT für bestimmte Basisblöcke (die teilweise nicht einmal innerhalb der selben Funktion wie der eingefügte Spillcode liegen) wesentlich niedrigere WCET-Werte berechnet. Die resultierende WCET ist daher für die dynamische Allokation in vielen Fällen deutlich geringer als die durch die statische Allokation minimal erreichte WCET. Dies ist in Abbildung 7.8 zu erkennen. Scratchpad-Größen, bei denen Objekte dynamisch kopiert werden, also Spillcode eingefügt wird, sind mit einem Pfeil markiert. Für alle Scratchpad-Größen, bei denen kein Spillcode erzeugt wird, entspricht das Resultat der dynamischen Allokation wie zu erwarten exakt dem der statischen.

Der maximale Gewinn einer Scratchpad-Allokation kann im Allgemeinen genau dann erzielt werden, wenn alle Datenobjekte über die gesamte Laufzeit des Programms im Scratchpad-Speicher liegen. Dies ist für den untersuchten Benchmark `ndes` ab einer Scratchpad-Größe von 1300 Bytes der Fall. Verglichen mit einer Scratchpad-Größe von 0 Bytes beträgt der Gewinn der statischen Allokation hier 23152 Zyklen. Auch durch die dynamische Allokation kann kein höherer Gewinn erzielt werden, da hier ebenfalls alle Datenobjekte von Beginn an im schnelleren Speicher liegen. Es ist also auch kein Spillcode erforderlich.

Betrachtet man die für eine Scratchpad-Größe von 1200 Bytes berechneten Werte, so fällt auf, dass durch die dynamische Allokation ein Gewinn erzielt werden kann, der höher ist als bei Auslagerung aller Datenobjekte (er beträgt 36823 Zyklen gegenüber einer Größe von 0 Bytes). Dies kann nicht durch eine effizientere Scratchpad-Nutzung erklärt werden, da die effizienteste Nutzung genau dann erfolgt, wenn alle Datenobjekte ausgelagert werden können. Die Ursache

für diesen höheren Gewinn liegt also bei dem eingefügten Spillcode. Die durch dessen Einfügen verursachten Änderungen in der Struktur des Codes scheinen das Programmverhalten in einer Weise zu beeinflussen, die eine deutlich schnellere Ausführung entlang des Worst-Case Execution Pfads ermöglicht.

Um zu zeigen, dass dieses Fehlverhalten tatsächlich nur durch den eingefügten Spillcode in Erscheinung tritt, wurde der Algorithmus so modifiziert, dass für jede Auslagerung lediglich der benötigte Spillcode erzeugt wird, jedoch keine Änderung der Zugriffsadressen vorgenommen wird. Das so resultierende Programm greift somit für alle Datenobjekte weiterhin auf den SRAM zu, der Scratchpad-Speicher wird nicht genutzt. Es wäre zu erwarten, dass in diesem Fall eine deutliche Verschlechterung der WCET festgestellt wird, da nicht von der Auslagerung profitiert werden kann, der Spillcode aber dennoch ausgeführt wird und entsprechende Kosten verursacht. Leider wird in vielen Fällen aber dennoch eine Verbesserung der WCET gemessen. Die für diesen Fall durch aiT berechneten WCET Werte sind als dritte Datenreihe in Abbildung 7.8 ersichtlich.

Betrachtet man die Werte bei einer Scratchpad-Größe von 100 Bytes wird deutlich, dass alleine durch das Einfügen von Spillcode die WCET deutlich gesenkt werden kann (um genau 16835 Zyklen). Dieser Gewinn ist sogar höher als der durch die statische Allokation für diese Größe erzielte (10808 Zyklen). Die so eingesparten Zyklen sind auch in dem für die dynamische Allokation berechneten Gewinn (30921 Zyklen) enthalten. Welcher Gewinn also tatsächlich durch die dynamische Auslagerung von Datenobjekten erreicht wird, kann nicht bestimmt werden. Es ist jedoch aufgrund der großen Unterschiede zwischen den Werten zu erwarten, dass auch ohne die beschriebene Timing-Anomalie bessere Ergebnisse als durch eine statische Auslagerung erzielt werden könnten.

Nur bei einer Scratchpad-Größe von 600 Bytes hat das Einfügen von Spillcode tatsächlich eine negative Auswirkung auf die WCET (sie verschlechtert sich um 848 Zyklen). In diesem Fall ist die WCET bei dynamischer Allokation dennoch besser als bei statischer Allokation. Der Unterschied beträgt 1300 Zyklen.

Eine Überprüfung der WCET der entsprechenden Spillcode-Basisblöcke ergab, dass die Kosten von diesen korrekt angerechnet wurden. Auch bei Ersetzung aller Befehle des Spillcodes durch `NOP`-Operationen, die keine Auswirkungen auf den Stack oder Speicher haben, tritt dieses Problem auf. Ein Fehler in der Implementierung des Spillcodes kann daher als Ursache ausgeschlossen werden.

Auch wenn in den Fällen, in denen eine dynamische Auslagerung von Datenobjekten bestimmt wurde, teilweise deutlich bessere Ergebnisse als durch die statische Allokation erzielt wurden, können diese nicht vollständig dem Erfolg der dynamischen Allokation zugeschrieben werden, sondern sie sind zu einem großen Anteil in den beschriebenen Timing-Anomalien begründet.

7.3.3 Fazit

Die Ursache der beschriebenen Anomalien, die beim Einfügen von Spillcode auftreten, konnte in der für diese Arbeit zur Verfügung stehenden Zeit leider nicht geklärt werden. Solange jedoch deren Einfluss auf die WCET nicht genau bestimmt werden kann, ist eine weitere Untersuchung der so erzielten Ergebnisse nicht sinnvoll, da nicht beurteilt werden kann, ob eine Verbesserung der WCET tatsächlich in der Optimierung begründet ist oder durch andere Nebeneffekte hervorgerufen wird.

Die in dieser Arbeit beschriebene dynamische Scratchpad-Allokation stellt dennoch einen vielversprechenden Ansatz dar. Für entsprechend angepasste Programme kann bereits jetzt eine dynamische Auslagerung von Datenobjekten berechnet werden. Die durch den Algorithmus getroffenen Entscheidungen erscheinen bei Betrachtung des Programmcodes in jedem Fall logisch und eine Verbesserung der WCET auch gegenüber der statischen Allokation ist sehr wahrscheinlich.

Leider konnte keine endgültige Aussage darüber getroffen werden, welche Gewinne sich auf diese Weise erzielen lassen. Eine weitere Untersuchung der beschriebenen Probleme und die Vermeidung der beschriebenen Einschränkungen erscheinen jedoch als lohnenswertes Ziel künftiger Arbeiten.

Kapitel 8

Zusammenfassung und Ausblick

Dieses Kapitel fasst zunächst die vorliegende Arbeit zusammen. Es wird erläutert, wie die im Vorfeld gesetzten Ziele erreicht wurden und welche Ergebnisse erzielt werden konnten.

In Abschnitt 8.2 wird schließlich ein Ausblick darauf gegeben, welche Forschungen und Entwicklungen auf Grundlage dieser Arbeit in Zukunft denkbar sind. Dabei wird auf Erweiterungs- und Verbesserungsmöglichkeiten der vorgestellten Algorithmen eingegangen und Anregungen zu weiteren datenzentrierten Optimierungen gegeben, die auf den im Rahmen dieser Arbeit entstandenen Erweiterungen des WCC aufbauen.

8.1 Zusammenfassung

Das Ziel dieser Diplomarbeit war die Entwicklung eines Algorithmus zur Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung. Dieser sollte in den am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelten C-Compiler WCC integriert werden.

Die vorliegende Arbeit verbindet also das inzwischen ausgiebig erforschte Gebiet der Scratchpad-Allokation mit dem relativ jungen Forschungszweig der WCET-Optimierung. Letzterer zeichnet sich durch eine Reihe von Merkmalen aus, die bei der Optimierung der durchschnittlichen Programmlaufzeit nicht von Bedeutung sind. Insbesondere die für eine gute Optimierung zwingend notwendige Betrachtung des instabilen Worst-Case Execution Pfads bildet hier eine große Herausforderung.

Um eine Grundlage für Verfahren zur Scratchpad-Allokation zu bilden, musste zunächst die Infrastruktur des WCC in zahlreichen Punkten erweitert werden. Erst so konnte ein Verschieben von Daten in verschiedene Speicher der Zielplattform ermöglicht werden. Bei diesen Erweiterungen wurde darauf geachtet, dass sie möglichst universell einsetzbar sind und auch von anderen Komponenten des WCC genutzt werden können. Für zukünftige Arbeiten, die sich mit Optimierungen durch Auslagerung oder Transformation von Daten beschäftigen, müssen somit keine großen Anpassungen an Compiler-Komponenten wie LLIR oder Code Selector mehr vorgenommen werden.

Für die Umsetzung der Optimierungen war weiterhin das Bereitstellen von Informationen über die Datenzugriffe der einzelnen Instruktionen bzw. Basisblöcke notwendig. Dieses wurde in einer

Form umgesetzt, die Zugriffe sowohl aus dem Programm selbst als auch aus der Analyse durch aiT erkennt. Dadurch können sehr genaue Informationen erhoben werden, die im Gegensatz zu Profiling-Ansätzen alle möglichen Programmabläufe berücksichtigen. Durch die Übergabe der im Code Selector gewonnenen Datenzugriffsinformationen an die WCET-Analyse kann zudem die WCET-Abschätzung verbessert werden, da aiT selbst nicht alle Datenzugriffe korrekt erkennen kann.

Zur Bestimmung der in den Scratchpad-Speicher auszulagernden Datenobjekte wurde ein ILP-basiertes Verfahren gewählt. Dies ermöglicht die implizite Betrachtung aller Kontrollflusspfade eines Programms und kann so bei Vernachlässigung der beschriebenen Einschränkungen optimale Ergebnisse erzielen. Da keine wiederholten Berechnungen des Worst-Case Execution Pfads nötig sind, ist die Laufzeit des Algorithmus auch für große Programme praxistauglich.

Eine Adaption der entwickelten Algorithmen auf andere Plattformen sollte leicht möglich sein, da sämtliche plattformabhängigen Daten lediglich Konstanten innerhalb des ILP darstellen. Für den TriCore Prozessor war insbesondere die Berechnung der Blockgewinne bei Auslagerung eines Datenobjekts eine Herausforderung, die durch eine vereinfachte Simulation der Load/Store-Pipeline gelöst wurde. Für weniger komplexe Systeme können hier feste Werte genutzt werden, so dass lediglich der Aufbau der ILP-Formulierung portiert werden müsste.

Die als Ziel dieser Arbeit gesetzte statische Allokation globaler und statischer Daten wurde vollständig umgesetzt und deren Qualität anhand zahlreicher praxisnaher Benchmarks geprüft. Dabei konnten für viele Programme große Einsparungen der WCET in der Größenordnung von bis zu 38% erzielt werden. Insbesondere konnte gezeigt werden, dass die durch den Algorithmus getroffenen Entscheidungen auch dann gute Resultate erzielen, wenn aufgrund beschränkter Größe nur eine kleine Teilmenge aller Datenobjekte in den Scratchpad-Speicher ausgelagert werden kann. Durch die vollständige Integration in den Compiler ist für den Nutzer kein weiterer Aufwand als die Angabe eines Kommandozeilenflags notwendig. Einem produktiven Einsatz der entwickelten Optimierung steht somit nichts im Wege.

Ein optionales Ziel der Arbeit war die Erweiterung des Algorithmus, so dass eine dynamische Allokation berechnet wird. Die Möglichkeit, den Scratchpad-Inhalt zur Laufzeit zu verändern, bringt eine ganze Reihe weiterer Herausforderungen, sowohl für die Formulierung des ILP als auch in der Umsetzung der so berechneten Allokation. Die ILP-Formulierung der statischen Allokation konnte in wesentlichen Punkten erweitert werden, um eine dynamische Auslagerung von Datenobjekten unter Berücksichtigung der auftretenden Kopierkosten zu ermöglichen. Der Ansatz, eine Veränderung des Scratchpad-Inhalts anstatt entlang der Kanten des Kontrollflussgraphen in verschachtelten Knoten zu ermöglichen, ist auf diesem Gebiet zuvor noch nicht untersucht worden.

Auch wenn in der derzeitigen Form noch einige Einschränkungen und Probleme bei der Bestimmung der so erzielten Ergebnisse den praktischen Einsatz der dynamischen Allokation verhindern, erscheint eine weitere Untersuchung dieses Ansatzes lohnenswert. Dessen Integration in den WCC wurde ebenfalls so weit implementiert, dass nur in wenigen Punkten noch Erweiterungen nötig sind.

Es konnten also weit über die Minimalziele dieser Arbeit hinausgehende Forschungsergebnisse erzielt werden, die zahlreiche Ansätze bilden, auf denen zukünftige Arbeiten aufbauen können. In welcher Form dies erfolgen kann, soll im nächsten Abschnitt behandelt werden.

8.2 Ausblick

Aufbauend auf dieser Arbeit sind eine ganze Reihe von zukünftigen Forschungsprojekten denkbar, die im Folgenden beschrieben werden sollen.

Die statische Scratchpad-Allokation konnte soweit abgeschlossen werden, dass hier lediglich Detailverbesserungen möglich sind. Es könnte beispielsweise auch die Auslagerung von lokalen Variablen, die typischerweise auf dem Stack gespeichert werden, ermöglicht werden. Da diese jedoch innerhalb der ILP-Fomulierung auf die gleiche Art und Weise wie globale Daten behandelt werden könnten, würde dies lediglich einigen Implementierungsaufwand erfordern, eine grundsätzliche Änderung des Algorithmus ist hierfür nicht notwendig. Im WCC ist zudem die Umsetzung einer solchen Erweiterung nicht direkt möglich, da hier der Stack bereits vollständig im Scratchpad-Speicher des TC1796 liegt.

Im Gegensatz dazu bietet der vorgestellte Ansatz zur dynamischen Allokation noch sehr viel Verbesserungspotential. Insbesondere die nötige Anpassung des Quellcodes von Programmen, deren Funktionen an unterschiedlichen Stellen aufgerufen werden, ist nicht wünschenswert. Der Versuch, die ILP-Formulierung dahingehend zu erweitern, dass in solchen Fällen keine Abhängigkeiten zwischen den aufrufenden Funktionen entstehen, erscheint daher als lohnenswertes Ziel anschließender Arbeiten. Eventuell können so sogar für eine Funktion je nach Kontext unterschiedliche Allokationen berechnet werden, um den Scratchpad-Speicher effizienter zu nutzen. Zumindest besteht jedoch die Möglichkeit, entsprechende Funktionen schon innerhalb des Compilers zu vervielfachen, so dass keine Nutzerinteraktion mehr notwenig ist. Dann ist jedoch zu untersuchen, welchen negativen Einfluss die Vergrößerung des resultierenden Programms hat.

Neben der Beseitigung der erwähnten Schwächen der ILP-Formulierung ist auch in deren Umsetzung noch einiger Aufwand nötig, um den praktischen Einsatz der dynamischen Allokation zu ermöglichen. Hier ist insbesondere die Anpassung der Zugriffsadressen von Instruktionen, die auf ausgelagerte Datenobjekte zugreifen, notwendig. Wie bereits beschrieben ist dies keine leichte Aufgabe, denn eine Adressberechnung kann sehr weit vor dem tatsächlichen Datenzugriff erfolgen und auch durch arithmetische Operationen zwischenzeitlich geändert worden sein. Es kann also vorkommen, dass die Adresse eines Datenobjektes in ein Adressregister geladen wird, bevor dieses in den Scratchpad-Speicher ausgelagert wird. In dem Register befindet sich dann die SRAM-Adresse dieses Objekts, und zum Zeitpunkt des Zugriffs würde so nicht auf den schnelleren Speicher zugegriffen werden. Außer, dass so kein Performancegewinn erzielt werden kann, ist auch die Korrektheit des Programms nicht mehr gewährleistet. Um dies zu verhindern, muss einiger Aufwand betrieben werden, um die in Adressregister geladenen Werte zu verfolgen, und unter Umständen Veränderungen an der Programmstruktur vorzunehmen.

Weitere Detailverbesserungen sind beim Einfügen von Spillcode möglich, da bisher in speziellen Fällen die Codegröße durch mehrmaliges Einfügen des selben Codes an unterschiedlichen Kanten unnötig vergrößert wird.

Insbesondere ist zu untersuchen, welche Ursachen für die teilweise großen Schwankungen der WCET beim Einfügen von zusätzlichem Code in ein Programm verantwortlich sind. Ohne eine Lösung dieses Problems kann keine Bewertung der durch die dynamische Optimierung erzielten Ergebnisse stattfinden. Es konnte festgestellt werden, dass dieses Problem unabhängig von dem in dieser Arbeit vorgestellten Algorithmus existiert, und daher potentiell auch weitere Optimierungen, die eine Änderung des Programmcodes vornehmen, davon betroffen sein können.

Ein weiterer Ansatz, auf dieser Arbeit aufzubauen, bietet sich in der Untersuchung von Programm-Transformationen, die den Erfolg der vorgestellten Optimierungen begünstigen. Dazu gehört beispielsweise das Aufsplitten von Arrays, die sonst zu groß für eine Einlagerung in den Scratchpad-Speicher wären. Im Rahmen der WCET-Optimierung könnten genau jene Bereiche eines Arrays ausgegliedert werden, auf die innerhalb des Worst-Case Execution Pfads zugegriffen wird. So könnten gerade bei kleinen Scratchpad-Größen noch bessere Ergebnisse erzielt werden.

Schließlich können die im Rahmen dieser Arbeit entwickelten Erweiterungen des WCC um eine Darstellung von Daten und Datenzugriffen auch als Grundlage ganz anderer Optimierungen genutzt werden. So könnte beispielsweise untersucht werden, ob durch die Umsortierung mehrdimensionaler Arrays und die Anpassung von darauf zugreifenden Schleifen eine WCET-Ersparnis erzielt werden kann.

Anhang A

Benchmark-Werte

	0	8	16	32	64	128	256	512
adpcm_decoder	555182	555126	555062	554960	554730	554480	554006	553670
adpcm_encoder	798522	798476	798424	798344	798160	797822	797210	796552
binarysearch	299	299	299	299	299	287	287	287
bsort100	318197	318197	318197	318197	318197	318197	318197	224147
compressdata	2789	2669	2611	2611	2581	2581	2581	2581
countnegative	51848	49848	49848	49848	49848	49848	49848	49848
crc	185295	185291	181713	181709	181709	181709	181709	181709
edge_detect	1899710	1899710	1879202	1879202	1838649	1838649	1838649	1838649
fdct	5791	5791	5791	5791	5791	5242	5242	5242
fft1	54528	54528	54528	54188	53780	53780	53780	53780
fir	12634	12634	12634	12634	12634	12634	11949	11949
g721_encode	1730668	1728876	1727340	1655088	1650992	1649968	1648944	1648944
g723_encode	1725070	1721230	1719694	1587538	1581394	1580626	1579346	1579346
gsm_encode	21596195	21596195	21596195	21596035	21594113	21581473	21581313	21580513
h263	10488155	10260511	10260511	10260511	10260511	10260511	10260511	10260511
hamming_window	70312	70312	70312	70312	70312	70312	65752	65752
insertsort	3050	3050	3050	3050	2403	2403	2403	2403
jfdctint	7014	7014	7014	7014	7014	7014	6438	6438
lcdnum	795	774	774	774	774	774	774	774
lms	1453652	1444808	1442195	1440984	1440984	1411437	1411437	1393548
ludcmp	9947	9947	9947	9947	9947	9947	9872	9842
matmult	600547	596547	596547	596547	596547	596547	596547	596547
md5	104456884	104202454	104202454	104202454	104202454	80297246	80297246	80297246
ndes	174984	168936	168929	168929	167651	163811	154888	151613
petrinet	8435	6957	6861	6019	5877	5877	5877	5877
qurt	8685	8673	8642	8612	8612	8612	8612	8612
searchmultiarray	41304	41304	41304	41304	41304	41304	41304	41304
selection_sort	4217551	4217551	4217551	4217551	4217551	4217551	4217551	4217551
st	500110	476098	466098	466098	466098	466098	466098	466098
statemate	304221	284324	274924	267932	256393	243478	239022	239022
test3	382079713	367971088	367971088	367971088	367971088	367971088	367971088	367971088

Tabelle A.1: Ergebnisse der statischen Allokation für SP-Größen von 0 bis 512 Bytes

	1024	2048	4096	8192	16384	32768	40960
adpcm__decoder	553642	553634	553634	553634	553634	553634	553634
adpcm__encoder	796468	796470	796470	796470	796470	796470	796470
binarysearch	287	287	287	287	287	287	287
bsort100	224147	224147	224147	224147	224147	224147	224147
compressdata	2581	2581	2581	2581	2581	2581	2581
countnegative	49848	46248	46248	46248	46248	46248	46248
crc	181463	181299	181299	181299	181299	181299	181299
edge__detect	1838649	1838649	1838649	1830098	1825921	1825921	1666765
fdct	5242	5242	5242	5242	5242	5242	5242
fft1	53780	53780	53780	53780	53780	53780	53780
fir	11949	11949	10824	10824	10824	10824	10824
g721__encode	1648944	1647921	1578972	1578972	1578972	1578972	1578972
g723__encode	1579346	1578323	1531901	1531901	1531901	1531901	1531901
gsm__encode	21504067	21503587	21503587	21503587	21503587	21503587	21503587
h263	10260511	10260359	10259314	10259314	10259309	10259297	10259240
hamming__window	65752	61192	57392	57392	57392	57392	57392
insertsort	2403	2403	2403	2403	2403	2403	2403
jfdctint	6438	6438	6438	6438	6438	6438	6438
lcdnum	774	774	774	774	774	774	774
lms	1393548	1392445	1392445	1392445	1392445	1392445	1392445
ludcmp	9842	9842	9842	9842	8916	8916	8916
matmult	596547	596544	596544	436547	436547	436547	436547
md5	80297246	80297246	80297246	80297246	80297246	80297246	80297246
ndes	151297	150653	150653	150653	150653	150653	150653
petrinet	5877	5877	5877	5877	5877	5877	5877
qurt	8612	8612	8612	8612	8612	8612	8612
searchmultiarray	41304	41304	38179	38175	38175	38175	38175
selection__sort	4217551	3857618	3857618	3857618	3857618	3857618	3857618
st	466098	466098	461098	456098	456098	456098	456098
statemate	239022	239022	239022	239022	239022	239022	239022
test3	367971088	367971088	348051581	312190914	270115735	238253968	236772678

Tabelle A.2: Ergebnisse der statischen Allokation für SP-Größen von 1024 bis 40960 Bytes

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[ABS02] Avissar, Oren ; Barua, Rajeev ; Stewart, Dave: An optimal memory allocation scheme for scratch-pad-based embedded systems. In: *ACM Transactions on Embedded Computing Systems (TECS)* 1 (2002), November, Nr. 1, S. 6–26

[Abs08a] AbsInt Angewandte Informatik GmbH: *aiT: Worst-Case Execution Time Analyzers*. `http://www.absint.com/ait`. Version: 2008

[Abs08b] AbsInt Angewandte Informatik GmbH (Hrsg.): *CRL Library Documentation*. AbsInt Angewandte Informatik GmbH, 2008. `http://www.absint.com/artist2/doc/crl2/html/index.html`

[Abs08c] AbsInt Angewandte Informatik GmbH (Hrsg.): *Worst-Case Execution Time and Stack Analysis aiT/StackAnalyzer for TriCore*. AbsInt Angewandte Informatik GmbH, August 2008

[AP01] Alevras, Dimitris ; Padberg, Manfred W.: *Linear Optimization and Extensions: Problems and Solutions*. Berlin : Springer, 2001

[Bih05] Bihr, Holger: *Entwicklung einer plattformunabhängigen, kontext-sensitiven Aliasanalyse für das ICD-C Compiler-Framework*. Dortmund, Universität Dortmund, Diplomarbeit, November 2005

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

[CP01] Colin, Antoine ; Puaut, Isabelle: A Modular & Retargetable Framework for Tree-Based WCET Analysis. In: *Proceedings of the 13th Euromicro Conference on Real-Time Systems (ECRTS)*. Delft, June 2001, S. 37

[cpl08] *ILOG CPLEX*. Website. `http://www.ilog.com/products/cplex/`. Version: 2008

[DP07] Deverge, Jean-Francois ; Puaut, Isabelle: WCET-Directed Dynamic Scratchpad Memory Allocation of Data. In: *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*. Pisa, Juli 2007, S. 179–190

[Erm03] Ermedahl, A.: *A Modular Tool Architecture for Worst-Case Execution Time Analysis*. Uppsala, Sweden, Uppsala University, Dept. of Information Technology, Diss., Juni 2003

[FH04] Ferdinand, Christian ; Heckmann, Reinhold: aiT: Worst Case Execution Time Prediction by Static Program Analysis. In: *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS)*. Santa Fe, April 2004

[FLT06] Falk, Heiko ; Lokuciejewski, Paul ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET)*. Schloss Dagstuhl : Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Juli 2006

[FS06] Falk, Heiko ; Schwarzer, Martin: Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In: *Proceedings of the 4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*. Seoul, Oktober 2006, S. 115–120

[GE99] Güting, Ralf H. ; Erwig, Martin: *Übersetzerbau: Techniken, Werkzeuge, Anwendungen*. Bd. 1. Berlin : Springer, 1999

[GNU08a] *GNU Binutils*. Website. `http://www.gnu.org/software/binutils/`. Version: 2008

[GNU08b] *GCC, the GNU Compiler Collection*. Website. `http://gcc.gnu.org`. Version: 2008

[ICD05] Informatik Centrum Dortmund (Hrsg.): *ICD-C Compiler Framework Developer Manual*. Informatik Centrum Dortmund, Januar 2005

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

[ICD08] *Informatik Centrum Dortmund Embedded Systems Profit Center*. Website. `http://www.icd.de/es/`. Version: August 2008

[Inf02] Infineon Technologies (Hrsg.): *TriCore 1.3 Architecture Overview Handbook*. V1.3.3. Infineon Technologies, Mai 2002

[Inf03] Infineon Technologies (Hrsg.): *TriCore Compiler Writer's Guide*. V1.4. Infineon Technologies, Dezember 2003

[Inf04a] Infineon Technologies (Hrsg.): *Memory Access Time in TriCore 1 TC1M Based Systems*. V1.1. Infineon Technologies, Juni 2004

[Inf04b] Infineon Technologies (Hrsg.): *TriCore 1 Pipeline Behaviour and Instruction Execution Timing*. V1.1. Infineon Technologies, Juni 2004

[Inf06] Infineon Technologies (Hrsg.): *TC1796 Data Sheet*. V0.7. Infineon Technologies, März 2006

[Inf08] Infineon Technologies (Hrsg.): *TriCore v1.3 Instruction Set*. V1.3.8. Infineon Technologies, Januar 2008

[ISO07] ISO/IEC (Hrsg.): *ISO/IEC 9899:1999 - Programming languages – C*. TC3. ISO/IEC, September 2007. `http://www.open-std.org/JTC1/SC22/WG14/www/docs/n1256.pdf`

[Jon05] Jones, M. T.: *Optimization in GCC.* Website. `http://www.linuxjournal.com/article/7269`. Version: Januar 2005

[Kel07] Keller, John: Developers of real-time embedded software take aim at code complexity. In: *Military and Aerospace Electronics* (April 2007). `http://mae.pennnet.com/articles/article_display.cfm?article_id=289158`

[Lee08] Lee, Corinna G.: *UTDSP Benchmark Suite.* Website. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`. Version: August 2008

[LLPM04] Lee, Sheayun ; Lee, Jaejin ; Park, Chang Y. ; Min, Sang L.: A Flexible Tradeoff Between Code Size and WCET Using a Dual Instruction Set Processor. In: *Proceedings of the 8th International Workshop on Software and Compilers for Embedded Systems (SCOPES).* Amsterdam, September 2004, S. 244–258

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

[LPMS97] Lee, Chunho ; Potkonjak, Miodrag ; Mangione-Smith, William H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. In: *Proceedings of the 30th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* North Carolina, Dezember 1997

[lps08] *lp_solve 5.5 Reference Guide.* Website. `http://lpsolve.sourceforge.net/`. Version: 2008

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

[MGCK07] Mutzel, Prof. Dr. P. ; Gutwenger, Carsten ; Chimani, Markus ; Klein, Karsten: *Skript zur Vorlesung Algorithm Engineering.* Universität Dortmund, 2007 `http://ls11-www.cs.uni-dortmund.de/people/gutweng/AE-07/komb_opt.pdf`

[MMSH01] Memik, Gokhan ; Mangione-Smith, William H. ; Hu, Wendong: NetBench: A Benchmarking Suite for Network Processors. In: *Proceedings of the 2001 International Conference on Computer-Aided Design (ICCAD).* San Jose, November 2001

[mrt08] *Mälardalen WCET research group benchmarks.* Website. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`. Version: August 2008

[PB00] Puschner, Peter ; Burns, Alan: A Review of Worst-Case Execution-Time Analysis. In: *Journal of Real-Time Systems* 18 (2000), Mai, Nr. 2/3, S. 115–128

[PND98] Panda, Preeti R. ; Nicolau, Alexandru ; Dutt, Nikil: *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration.* Norwell : Kluwer Academic Publishers, 1998

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

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

[Sta07] Statistisches Bundesamt Deutschland: *Fast 70 Prozent der Bevölkerung ab zehn Jahren nutzen das Internet.* Website. `http://www.destatis.de/jetspeed/portal/cms/Sites/destatis/Internet/DE/Presse/pm/2007/11/PD07__486__63931,templateId=renderPrint.psml`. Version: 2007, Abruf: 13.06.2008

[Sta08] *Telelogic Statemate.* `http://modeling.telelogic.com/products/statemate/index.cfm`. Version: 2008

[SWLM02] Steinke, S. ; Wehmeyer, L. ; Lee, B. ; Marwedel, P.: Assigning Program and Data Objects to Scratchpad for Energy Reduction. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE).* Paris, März 2002

[UB03] Udayakumaran, Sumesh ; Barua, Rajeev: Compiler-decided Dynamic Memory Allocation for Scratch-pad Based Embedded Systems. In: *Proceedings of the 2003 International Conference on Compilers, Architecture and Synthesis for embedded systems (CASES).* San Jose, Oktober/November 2003, S. 276–286

[UDB06] Udayakumaran, Sumesh ; Dominguez, Angel ; Barua, Rajeev: Dynamic Allocation for Scratch-pad Memory Using Compile-time Decisions. In: *ACM Transactions on Embedded Computing Systems (TECS)* 5 (2006), Mai, Nr. 2, S. 472–511

[VM06] Verma, Manish ; Marwedel, Peter: Overlay Techniques for Scratchpad Memories in Low Power Embedded Processors. In: *IEEE Transactions on Very Large Scale Integration Systems (VLSI)* 14 (2006), August, Nr. 8, S. 802–815

[VWM04] Verma, Manish ; Wehmeyer, Lars ; Marwedel, Peter: Dynamic Overlay of Scratchpad Memory for Energy Minimization. In: *Proceedings of the 2nd International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS).* Stockholm, September 2004, S. 104–109

[Weg03] Wegener, Ingo: *Komplexitätstheorie.* Berlin : Springer, 2003

[WM95] Wulf, Wm. A. ; McKee, Sally A.: Hitting the Memory Wall: Implications of the Obvious. In: *Computer Architecture News* 23 (1995), Nr. 1, S. 20–24

[WM04] Wehmeyer, Lars ; Marwedel, Peter: Influence of Onchip Scratchpad Memories on WCET prediction. In: *Proceedings of the 4th International Workshop on Worst-Case Execution Time Analysis (WCET).* Catania, Juni 2004

[WM05] Wehmeyer, Lars ; Marwedel, Peter: Influence of Memory Hierarchies on Predictability for Time Constrained Embedded Software. In: *Proceedings of the Conference on Design, Automation and Test in Europe (DATE).* München, März 2005

[WMZC07] Wei, Tao ; Mao, Jian ; Zou, Wei ; Chen, Yu: A New Algorithm for Identifying Loops in Decompilation. In: *Proceedings of the 14th International Static Analysis Symposium (SAS).* Kongens Lyngby, August 2007, S. 170–183

[ZWHM05] Zhao, Wankang ; Whalley, David ; Healy, Christopher ; Mueller, Frank: Improving WCET by applying a WC code-positioning optimization. In: *ACM Transactions on Architecture and Code Optimization (TACO)* 2 (2005), Dezember, Nr. 4, S. 335–365