TECHNISCHE UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Igor Ionov

Design und Realisierung von Konzepten fur retargierbare, multikriterielle Optimierungen im WCET-fähigen Compiler

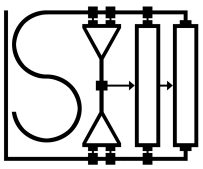
Diplomarbeit

12. März 2010

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl XII (Embedded System Design)
Fakultät für Informatik
Technische Universität Dortmund

Betreuer:
Prof. Dr. Peter Marwedel
Dipl.-Inf. Sascha Plazar



GERMANY · D-44221 DORTMUND

Vorwort

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

Mein Dank gilt hierbei vor allem meinem Betreuer Sascha Plazar, der mich durch seine ausdauernde Hilfe und Motivation stets bestmöglich unterstützt hat. Er hat sich immer Zeit für mich genommen, wenn ich Fragen hatte, und hat maßgeblich zur Erstellung dieser Arbeit beigetragen.

Außerdem bedanke ich mich bei allen anderen Mitarbeitern des Lehrstuhls 12 für Eingebettete Systeme, insbesondere Paul Lokuciejewski und Timon Kelter für ihre freundliche Unterstützung.

Schließlich danke ich auch meiner Familie dafür, dass sie mich stets bei meinem Studium unterstützt hat.

Inhaltsverzeichnis

1. Einleitung **5**
- 1.1. Motivation . . . 5
- 1.2. Ziele der Diplomarbeit . . . 8
- 1.3. Verwandte Arbeiten . . . 9
- 1.4. Aufbau der Diplomarbeit . . . 11

2. Grundlagen **13**
- 2.1. Zielplattformen . . . 13
  - 2.1.1. Die TriCore-Plattform . . . 13
  - 2.1.2. Die ARM-Plattform . . . 15
- 2.2. Optimierende Compiler . . . 17
  - 2.2.1. GCC . . . 19
  - 2.2.2. encc . . . 22
  - 2.2.3. WCC . . . 25

3. Retargierbares Compiler-Backend **33**
- 3.1. Grundlagen . . . 33
- 3.2. Integration einer Plattform . . . 35
  - 3.2.1. LLIR . . . 36
  - 3.2.2. GCC als Codeselector . . . 39
  - 3.2.3. GCC2LLIR . . . 41

4. Multikriterielle Optimierungen **43**
- 4.1. Grundlagen . . . 44
  - 4.1.1. Bestimmung von schwach-dominierenden Lösungen . . . 46
  - 4.1.2. Klassische Methoden . . . 47
  - 4.1.3. Evolutionäre Algorithmen . . . 49
- 4.2. Darstellung der Kriterien im Compiler . . . 51
  - 4.2.1. WCET . . . 52
  - 4.2.2. ACET . . . 56
  - 4.2.3. Energie . . . 61
  - 4.2.4. Codegröße . . . 64

5. Evaluierung **65**
- 5.1. Verwendete Optimierungen . . . 65

5.2. Vergleich der Einflüsse von Standardoptimierungen auf die ACET der Zielarchitekturen 67
5.2.1. Vergleich der Standardoptimierungsstufen 67
5.2.2. Vergleich der einzelnen Optimierungen 68
5.3. Vergleich der Einflüsse von Standardoptimierungen auf ACET und Energieverbrauch 73
5.3.1. Vergleich der Standardoptimierungsstufen 73
5.3.2. Vergleich der einzelnen Optimierungen 74
5.4. Fazit 78

6. Zusammenfassung / Ausblick **79**
6.1. Zusammenfassung 79
6.2. Ausblick 80

A. Verwendete Benchmarks **83**

B. Messergebnisse **85**

Literaturverzeichnis **93**

1. Einleitung

Dieses Kapitel soll in die vorliegende Diplomarbeit einführen. Abschnitt 1.1 motiviert das Themengebiet der Diplomarbeit. Im darauf folgenden Abschnitt 1.2 werden die Ziele beschrieben, die in dieser Diplomarbeit erreicht werden sollen. Im Abschnitt 1.3 wird ein kurzer Überblick über die bereits zu diesem Thema veröffentlichen Publikationen gegeben. Abschließend wird in Abschnitt 1.4 der Aufbau der Diplomarbeit erläutert.

1.1. Motivation

Bis zum Ende der achtziger Jahre war der Begriff der Datenverarbeitung stark mit großen Mainframe-Rechnern und riesigen Rechenzentren verbunden. Durch verbesserte Herstellungsverfahren und damit zusammenhängenden Verkleinerung der Chips und Peripheriegeräte hat sich der Anwendungsbereich von informationsverarbeitenden Systemen hin zu kleineren *Personal Computer*, den PCs, verlagert. Dieser Trend zur Miniaturisierung hat sich fortgesetzt, so dass die Mehrzahl der heutzutage verwendeten Informationssysteme in kleinen, meist mobilen Geräten integriert ist. Bereits im Jahr 2000 waren nach Schätzungen mehr als 90% der eingesetzten Prozessoren in den so genannten *eingebetteten Systemen* wiederzufinden [Mar07].

Solche eingebettete Systeme sind in der Regel informationsverarbeitende Systeme, die in ein größeres Produkt integriert sind und bei denen für den Benutzer nicht mehr die Funktion als Rechner im Vordergrund steht, sondern die konkrete Anwendung des Produkts [Mar07]. Dazu gehören Hard- und Softwaresysteme, die z.B. in Telekomminukationsgeräten, MP3-Playern, Autos oder Haushaltsgeräten eingesetzt werden. So sind in einem modernen Mittelklassewagen über 70 unterschiedliche Prozessoren eingebaut, die Software-Komponenten mit über einer Million Zeilen Quellcode beinhalten, Tendenz weiter steigend [Kel07].

Durch die enge Integration von eingebetteten Systemen in den Alltag eröffnen sich neue Anwendungsbereiche für informationsverarbeitende Systeme. Diese neue Art von Anwendungen wird auch *ubiquitouis computing* oder *ambient intelligence* genannt. Ubiquitos Computing beschreibt das Ziel, Informationen überall und für jedermann zugänglich zu machen, indem z.B. mobile Geräte den Zugriff auf Informationen möglich machen. Ambient Intelligence soll dagegen zur Steigerung der Lebensqualität jedes Einzelnen beitragen, ohne dass die entsprechende Person mit einem Informationssystem interagieren muss. Diese Kategorie der Informationssysteme wird hauptsächlich bei der Kommunikationstechnologie im Wohnbereich sowie im Bereich der „intelligenten“ Häuser eingesetzt, die z.B. das Raumklima selbständig regeln oder die Beleuchtung ohne das Zutun des Benutzers anpassen. Alle

oben aufgeführten Anwendungen wurden erst mit der großflächigen Verbreitung der eingebetteten Systeme ermöglicht, da erst diese Systeme den Forderungen nach notwendigen Mobilität und Größe gerecht werden.

Die Anwendungsgebiete der eingebetteten Systeme stellen besondere Anforderungen an die Entwickler. Gerade durch die Mobilität, die eine wichtige Eigenschaft vieler eingebetteter Systeme darstellt, kommen bestimmte Beschränkungen in Spiel, die beim Entwurf klassischer Softwaresysteme nur wenig oder keine Relevanz aufweisen. Um allen Anforderungen unter Beachtung der Einsatzumgebung gerecht zu werden, müssen eingebettete Systeme sehr effizient sein. Die Effizienz lässt sich dabei unter Anderem an folgenden Größen beurteilen:

1. *Energie*: Viele eingebettete Systeme sind in tragbare Geräte integriert, die ihre Stromversorgung über Batterien oder Akkus sicherstellen. Moderne Multimedia-Anwendungen, welche heutzutage vermehrt in solchen Geräten zum Einsatz kommen, haben hohe Anforderungen an die Rechenkapazität der unterliegenden Hardware. Diese Anforderungen lassen sich oft nur über Einsatz von leistungsstärkeren Prozessoren erfüllen, die im Allgemeinen mehr Energie verbrauchen als entsprechend langsamere Prozessoren. Um dennoch für Kunden akzeptabel lange Batterielaufzeiten zu gewährleisten, muss die vorhandene Energie sehr effizient eingesetzt werden.

2. *Codegröße*: Die auf einem eingebetteten System ausgeführte Software muss im System selbst abgespeichert sein. Der dazu verfügbare Speicherplatz ist meistens sehr begrenzt, da viele eingebettete Systeme anders als bei z.B. Desktop-Systemen keine Festplatten besitzen, sondern vergleichsweise kleine Flash-Speicher. Vor allem bei Ein-Chip-Systemen (engl. *System on a Chip*, Abk. *SoC*), die alle Elemente der informationsverarbeitenden Hardware auf einem Chip integrieren, ist der verfügbare Speicher implementierungsbedingt nur sehr begrenzt. Damit muss die für ein System vorgesehene Software den vorhandenen Speicherplatz zwangsläufig sehr effizient nuzten.

3. *Laufzeit-Effizienz*: Eingebettete Systeme müssen oft Echtzeitschranken einhalten. Die Korrektheit eines Programms ist damit nicht nur von den Ergebnissen der Berechnung abhängig, sondern auch von der Zeit, in der diese geliefert werden [BW01]. Dabei wird zwischen *harten* und *weichen* Echtzeitsystemen unterschieden. Zu Ersteren gehören Steuergeräte aus sicherheitskritischen Bereichen wie Autos, Flugzeugen, Kraftwerken oder ähnlichen industriellen Überwachungsanlagen. In solchen Systemen soll die Einhaltung der vorgegebenen Reaktionszeiten immer gewährleistet sein, da sonst Systemfehler leicht zu Katastrophen führen können. Dem gegenüber kommt bei weichen Echtzeitsystemen wie z.B. bei Video- oder MP3-Player bei einer Überschreitung der Zeitschranken zwar niemand zur Schaden, trotzdem können wiederholte Aussetzer für den Benutzer sehr störend sein. Neben der Korrektheit der Ergebnisse wird so die Kenntnis über die maximale Ausführungszeit zu einer wichtigen Voraussetzung bei der Entwicklung von Software für eingebettete Systeme.

4. *Gewicht*: Bei allen tragbaren Geräten wünscht sich der Benutzer, dass diese möglichst klein und leicht sind, um nicht zur Last zu fallen. Somit wird das Gesamtgewicht des Systems zum einem weiteren Kaufkriterium, das bei der Entwicklung beachtet werden muss. Auch hier gilt im Allgemeinen, dass kleinere Bauteile eher die leistungsschwächeren sind. Somit muss, bezogen auf die Anforderungen an das Gesamtsystem, ein Trade-Off zwischen den unterschiedlichen Kriterien gefunden werden.

Um schnelle Ausführungszeit und einen geringen Energieverbrauch, sowie eine geringe Größe der verbauten Systemkomponenten bei einem geringem Gewicht zu erreichen, müssen sowohl Hardware als auch Software sehr stark optimiert werden. Hardwareoptimierungen lassen sich oft nur über Einsatz von leistungsstärkeren Bauteilen realisieren, die entweder teurer sind oder einen größeren Energieverbrauch und Baugröße aufweisen. Daher muss zwangsläufig auch die Software für eingebettete Systeme stark optimiert werden. Wegen der Komplexität der heutigen Systeme und einer großen Vielfalt der auf dem Markt verfügbaren Prozessoren wird die Softwareentwicklung in einer Hochsprache, vorwiegend C betrieben und von einem Compiler für die Zielarchitektur übersetzt. Moderne Compiler bieten eine breite Auswahl an Optimierungen mit dem Ziel, die durchschnittliche Ausführungszeit (engl. *Average Case Execution Time*, Abk. *ACET*) [Leu00] oder neuerdings auch den Energieverbrauch [SWV10] zu minimieren. Dagegen sind Compiler-Optimierungen, die auf eine Minimierung der maximalen Ausführungszeit (engl. *Worst Case Execution Time*, Abk. *WCET*) abzielen bis heute relativ wenig erforscht. Solche Optimierungen basieren in der Regel auf der Integration eines statischen WCET-Analysetools in das Compilerframework, das Laufzeitinformationen zu Verfügung stellt, auf deren Basis dann die WCET-Optimierung eines Programms erfolgt. Zusammenhänge und Auswirkungen von unterschiedlichen Optimierungen auf mehrere Kriterien sind ebenfalls sehr unzureichend untersucht.

Meistens sind solche optimierende Compiler-Frameworks nur für einzelne Zielarchitekturen entwickelt. Solche strikte Bindung an bestimmte Prozessoren bringt mehrere Nachteile mit sich. Erstens kann die mit dem Compiler übersetzte Software nur auf dem unterstützten Prozessor laufen, was die Verwendung anderer Plattformen ausschließt. Zweitens kann jede entwickelte Optimierung nur auf einer einzigen Plattform ausreichend getestet werden, somit sind Aussagen über die Effizienz auf anderen Prozessoren schwierig bis gar unmöglich. Dabei können Optimierungen, die bestimmte Eigenschaften einer Architektur ausnutzen und deswegen eine Verbesserung für das jeweilige Kriterium bewirken auf einer anderen Architektur sich negativ auswirken. Drittens können die Compiler-Frameworks, die nur einen einzelnen Prozessor unterstützten nicht von Synergien in der Entwicklung von Optimierungen profitieren. Jedes Softwaremodul, welches bestimmte Optimierungen auf dem unterstützten Prozessor implementiert muss in Teilen oder komplett umgeschrieben werden, wenn es auf einen anderen Compiler mit anderer Zielarchitektur portiert werden soll. Dagegen kann ein Framework, das mehrere Prozessoren unterstützt die Entwicklungszeit für prozessorspezifischen Optimierungen enorm verkürzen, da generische Module wiederverwendet werden können. Auch können die von einem solchen Framework bereitgestellten High-Level Analysen und Optimierungen direkt für alle unterstützten Architekturen be-

nutzt werden.

Wenig Beachtung bei der Entwicklung eines Compiler-Frameworks hat bis jetzt auch die Wechselwirkung verschiedener Optimierungen auf mehrere unterschiedliche Kriterien gefunden. Gar nicht untersucht z.B. ist der Einfluss von energieminimierenden Optimierungen auf die WCET eines Programms. Die Kenntnis solcher Wechselwirkungen ist aber für die Erstellung von effizienter Software für eingebettete Systeme heutzutage unerlässlich. Somit soll mit dieser Diplomarbeit ein neuer Ansatz für Compiler-Frameworks entwickelt werden, der sowohl Unterstützung für mehrere Architekturen anbietet als auch die Auswirkungen verschiedener Optimierungen auf mehrere unabhängige Kriterien beachten soll.

1.2. Ziele der Diplomarbeit

Im Rahmen der Forschungsarbeit des Lehrstuhls Informatik 12 der Technischen Universität Dortmund ist mit dem WCC (WCET-Aware C Compiler) ein WCET-optimierender Compiler für den Infineon TriCore Prozessor entwickelt worden [FLT06, Lok07]. Der WCC ermittelt während der Übersetzung des Quellcodes vollautomatisch die obere Laufzeitschranke eines Programms, damit diese für die weiteren Optimierungen zur Verfügung steht. Dafür wird das WCET-Analysetool *aiT* von der Firma AbsInt Angewandte Informatik GmbH eingesetzt [Abs10].

Da der WCC in seiner jetzigen Form allerdings nur WCET und ACET als Optimierungskriterien unterstützt und nur für ein bestimmtes Zielsystem - dem Infineon TriCore Code generieren kann, soll das Compiler-Framework in dieser Diplomarbeit erweitert werden. Dabei soll sowohl Unterstützung für weitere Optimierungskriterien geschaffen werden, als auch Erweiterungen zur Codegenerierung für weitere Zielplattformen durchgeführt werden.

Als erstes wird das Compiler-Framework für ein anderes System retargiert. Dabei soll ein retargierbares Compiler-Backend entstehen, der für unterschiedliche Zielarchitekturen mit relativ wenig Aufwand angepasst werden kann. Dazu werden die TriCore-spezifischen Bestandteile des Compiler-Frameworks durch generische Komponenten für das entsprechende System ersetzt, wie z.B. der Codeselector durch einen Compiler für das Zielsystem. Diese Änderungen werden am Beispiel des ARM7TDMI-Prozessors realisiert.

Des Weiteren sollen Strukturen zur Entwicklung von multikriteriellen Optimierungen im WCC implementiert werden. Es sollen in dieser Diplomarbeit keine Optimierungen entwickelt, sondern vielmehr die Voraussetzungen für eine solche Entwicklung geschaffen werden. Das soll über Anbindung unterschiedlicher Analyse-Tools an das Compiler-Framework geschehen, welche die Auswirkungen von High-Level Optimierungen auf die einzelnen Kriterien untersuchen und für weitere Optimierungen zu Verfügung stellen. Unter Anderem sollen folgende Kriterien untersucht werden können:

- *WCET*: Die WCET eines Programms wird über die Anbindung von *aiT* für den TriCore bereits berechnet. Die Grundlage für eine solche Anbindung soll auch für die ARM-Architektur geschaffen werden.

- *ACET*: Die Berechnung der ACET ist noch kein Bestandteil des WCC und soll für alle Architekturen verfügbar gemacht werden. Das soll über Anbindung des Entwicklungs- und Simulationswerkzeugs *CoMET* der Firma VaST Systems [vas10] erfolgen.

- *Energie*: Die Berechnung des Energieverbrauchs eines Programms soll mithilfe des Analysetools *enProfiler* erfolgen, das als Bestandteil des energieoptimierenden Compilers encc am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelt wurde. Da der enProfiler nur ein Energiemodell für die ARM-Plattform enthält, soll die Betrachtung der Energie auch vorerst nur für diese Architektur implementiert werden. Die entwickelten Techniken zur Messung des Energieverbrauchs werden allerdings so angepasst, dass sie auch leicht für andere Architekturen benutzt werden können, lediglich das Energiemodell für die entsprechende Architektur muss erstellt werden.

Abschließend soll eine Evaluierung des Einflusses vom vorhandenen ACET-Standardoptimierungen im WCC auf die unterstützten Kriterien durchgeführt werden. Dazu werden die einzelnen Optimierungen auf eine Vielzahl von Programmen aus diversen Benchmark Suiten angewendet, und anschließend wird der Einfluss auf einzelne Kriterien ausgewertet. Dieses Wissen kann später für die Entwicklung von Optimierungen, die mehrere Kriterien gleichzeitig unterstützen, ausgenutzt werden.

1.3. Verwandte Arbeiten

In diesem Abschnitt soll ein kurzer Überblick über die bisher veröffentlichten Forschungsarbeiten gegeben werden, die sich mit ähnlichen Themen beschäftigt haben.

Das Thema der retargierbaren Compiler für eingebettete Systeme, die bei Programmübersetzung mehrere Optimierungskriterien beachten sollten, ist nicht neu. Erste Forderungen an Eigenschaften, die solche Compiler besitzen sollten wurden bereits in [Mar95] diskutiert. Allerdings muss feststellt werden, dass bis heute kein einziger Compiler alle diese Anforderungen in einem Framework realisiert.

Einen Überblick über die Entwicklung von retargierbaren Compiler wird in [Leu97] gegeben. Der hier beschriebene RECORD-Compiler zeigt für die Mehrzahl der Benchmarks bessere Ergebnisse als die zu dem Zeitpunkt verfügbaren plattformspezifischen Compiler. Dieses Beispiel zeigt, dass auch retargierbare Compiler sehr effizienten Code produzieren können.

Relativ gut erforscht sind Compiler-Optimierungen für einzelne Kriterien wie Energie oder ACET, die bei Codegenerierung für einen Prozessor eingesetzt werden. In [WHM04] werden Techniken vorgestellt, die mit Hilfe von ganzzahliger Linearer Programmierung den Energieverbrauch von Systemen mit mehreren Scratchpad-Speichern um bis zu 22% senken können, indem Speicherobjekte während der Kompilierung optimal auf unterschiedliche Partitionen des Scratchpads-Speicher verteilt werden. Zur Berechnung des Energieverbrauchs wurde in dieser Arbeit das Analysetool *enProfiler* entwickelt, welches auch in dieser Diplomarbeit eingesetzt werden soll.

Eine breite Betrachtung der Methoden zur ACET-Minimierung bei Generierung von Code für eingebettete Systeme ist auch in [Leu00] oder allgemein für alle Architekturen in [SS03] nachzulesen. Unter Anderem werden hier quellsprachen- und architekturunabhängige Optimierungen auf SSA-Bäumen vorgestellt. Weitere hier beschriebene Optimierungen basieren auf Laufzeitinformationen, die während der Ausführung des nicht optimierten Programms gewonnen werden. Ausgehend von diesen Laufzeitinformationen können die Optimierungen zielgerichtet an den Stellen des Quellcodes angewendet werden, die am meisten zur Programmlaufzeit beitragen. Ein ähnlicher Ansatz wird auch in dieser Diplomarbeit verwendet, um spätere Entwicklung von Optimierungen für weitere Kriterien zu ermöglichen.

Im Gegensatz zu den bisher vorgestellten Arbeiten sind Compiler-Optimierungen, die auf eine Minimierung der WCET abzielen bis heute relativ wenig erforscht. Alle Ansätze basieren auf einer Interaktion zwischen dem Compiler und einem statischen WCET-Analysetool, das die WCET-Werte des Programms berechnet. Neben den unterstützten Prozessoren stellt die interne Repräsentation des Quellcodes das wichtige Unterscheidungsmerkmal dar. Fast alle solche Compiler-Frameworks arbeiten ausschließlich auf einer Low-Level Zwischendarstellung (engl. *Intermediate Representation*, Abk. *IR*), was die Anwendung von High-Level Optimierungen ausschließt.

Als Beispiel für einen solchen WCET-optimierenden Compiler kann das interaktive Compiler-System *VISTA* [ZKW+04] genannt werden. Dieser Compiler übersetzt C-Quellcode in eine interne Low-Level IR, auf welcher dann Code-Optimierungen ausgeführt werden. Durch Anbindung eines propriäteren statischen WCET-Analysetools, das zwei einfache Prozessoren ohne Caches unterstützt, können die ermittelten WCET-Informationen für WCET-minimierende Low-Level Optimierungen wie z.B. *block reordering* [ZWHM04] benutzt werden. Konstruktionsbedingt kann VISTA keine High-Level Optimierungen anwenden.

Ein weiteres Beispiel für einen WCET-minimierenden Ansatz stellt das open-source Tool *Heptane* [CP01] dar. Es ist ein WCET-Analysetool mit Unterstützung mehrerer Zielarchitekturen. Als Eingabe verarbeitet es C-Quellcode, der in eine High-Level IR übersetzt wird. Danach wird der Code in eine Low-Level IR transformiert, auf der die WCET-Analyse durchgeführt wird. Heptane wird ausschließlich für die Entwicklung von Low-Level Optimierungen wie z.B. *Predictable Page Allocations* [HP08] eingesetzt.

Im Gegensatz zu den beiden vorherigen Beispielen, die WCET-Optimierungen ausschließlich auf der Low-Level Ebene durchführen, kann der *WCET-Aware C Compiler* (Abk. *WCC*) Optimierungen auf der High-Level IR ausführen. Dieser Compiler wird ausführlich in Abschnitt 2.2.3 behandelt. Die Erweiterungen an dem WCC-Framework, die in dieser Diplomarbeit durchgeführt werden sollen, basieren auf [PLM08]. Anders als in dieser Publikation soll allerdings nicht das gesamte Backend des Compilers ersetzt werden, sondern nur an die jeweilige Architektur angepasst, was eine einfachere Integration der Analysetools und die Rücktransformation der Ergebnisse in die High-Level IR erleichtert. Auch werden Low-Level Optimierungen für unterschiedliche Architekturen erst auf diese Weise möglich.

In der Diplomarbeit von Fatih Gedikli [Ged08] wird die Transformation von WCET-Informationen für die Nutzung in High-Level Optimierungen beschrieben. Dieser Mechanismus wird unter Anderem in [LGM09] für die Entwicklung eines Verfahrens zur Beschleunigung von WCET-Optimierungen ausgenutzt.

1.4. Aufbau der Diplomarbeit

Zum Abschluss soll in diesem Abschnitt ein Überblick über die Gliederung dieser Diplomarbeit gegeben werden.

In Kapitel 2 sollen die Grundlagen, auf die im gesamten Verlauf der Diplomarbeit zurückgegriffen wird, eingeführt werden. Zuerst wird ein Überblick über das TriCore-Architektur gegeben, welche aktuell vom WCC unterstützt wird, sowie über die ARM-Architektur, die als Grundlage für das neue retargierbare Compiler-Backend dienen soll. Anschließend sollen die Besonderheiten, die bei der Entwicklung von Compilern für eingebettete Systeme zu beachten sind, dargestellt werden. Des Weiteren wird hier ein Überblick über einige aktuelle Compiler-Frameworks gegeben, mit einem Schwerpunkt auf dem Compiler-Framework des WCC, der eine Grundlage dieser Diplomarbeit darstellt.

Das Kapitel 3 beschäftigt sich mit den Änderungen am Framework, die für die Portierung des Compilers auf eine andere Architektur notwendig sind. Dafür wird eine kurze Übersicht über die sonst üblichen Methoden der Entwicklung von retargierbaren Compiler gegeben. Der Rest des Kapitels beschreibt die eigentliche Realisierung des Backends.

In Kapitel 4 werden die neuen Kriterien, die der WCC bei Übersetzung der Programme beachten soll, vorgestellt. Die Strukturen, die zur Unterstützung der Optimierungen für diese neuen Kriterien entwickelt wurden, werden gemeinsam mit den Analysetools für das jeweilige Kriterium beschrieben.

In Kapitel 5 erfolgt die Evaluierung des Einflusses von vorhandenen Standardoptimierungen des Compiler-Frameworks auf die vom WCC unterstützten Kriterien ACET und Energie. Dabei werden die einzelnen Optimierungen mithilfe unterschiedlicher Benchmarks für jedes einzelne Kriterium ausgeführt und miteinander verglichen. Des Weiteren wird ein Vergleich zwischen der TriCore- und der ARM-Architektur ausgewertet. Auch dazu werden mehrere Benchmarks eingesetzt.

Das Kapitel 6 fasst die Ergebnisse dieser Arbeit kurz zusammen und gibt einen Ausblick auf zukünftige Entwicklungsmöglichkeiten, die durch die Erweiterung des Frameworks geschaffen wurden.

2. Grundlagen

In diesem Kapitel sollen die Grundlagen beschrieben werden, auf die im weiteren Verlauf der Diplomarbeit zurückgegriffen wird. Dazu werden im Abschnitt 2.1 die verwendeten Zielplattformen eingeführt. Im Abschnitt 2.2 werden anschließend die Grundlagen von optimierenden Compiler für eingebettete Systeme dargestellt.

2.1. Zielplattformen

Unter einer Zielplattform wird der Prozessor verstanden, auf dem die durch den Compiler übersetzten Programme ausgeführt werden sollen. Im Folgenden werden die beiden Architekturen vorgestellt, die im Laufe dieser Diplomarbeit eingesetzt werden. Abschnitt 2.1.1 beschreibt den Infineon TriCore, der als ursprüngliche Zielplattform bei der Entwicklung des WCC verwendet wurde. Im Abschnitt 2.1.2 wird die ARM-Architektur vorgestellt, die als Zielarchitektur für die Realisierung eines retargierbaren Compiler-Backends in den WCC integriert wurde.

2.1.1. Die TriCore-Plattform

Die ursprünglich von dem WCC unterstützte Zielplattform ist der TriCore TC1796 [Inf07] von Infineon. Der vom Compiler generierte Code ist ebenfalls mit dem Nachfolgemodell TC1797 [Inf09] kompatibel. Die von Infineon entwickelte TriCore-Architektur stellt eine Reihe von Prozessoren dar, die für einen Einsatz in der Mess-, Regel-, Steuerungs- oder Automatisierungstechnik für eingebettete Systeme im Bereich Automotive beworben werden. Der Name TriCore ist von der 3-in-1-Architektur des Chips abgeleitet, die aus einem 32-Bit Mikrocontroller mit einer so genannten reduzierten Befehlssatz-Architektur (engl. *Reduced Instruction Set Computing*, Abk. *RISC*) sowohl komplexen Befehlssatz-Architektur (engl. *Complex Instruction Set Computing*, Abk. *CISC*) und erweiterter DSP-Funktionalität (engl. *Digital Signal Prozessor*, Abk. *DSP*) besteht. Dadurch soll bei relativ klein bleibender Größe des Chips die Gesamtleistung des Systems maximiert werden. Der Aufbau des TC1796-Prozessors, für den der WCC Code erzeugen kann, ist in der Abbildung 2.1 in einem Blockdiagramm dargestellt.

Der RISC-Prozessor besitzt eine 32-Bit Load/Store Harward-Architektur mit getrenntem Daten- und Instruktionsspeichern, 16 Adress- und 16 Datenregistern, und superskalare Ausführung mit drei vierstufigen Pipelines. Jeweils eine Pipeline ist für arithmetische Berechnungen, Load-/Store-Speicherzugriffe bzw. für schnelle Schleifenabarbeitung zuständig.

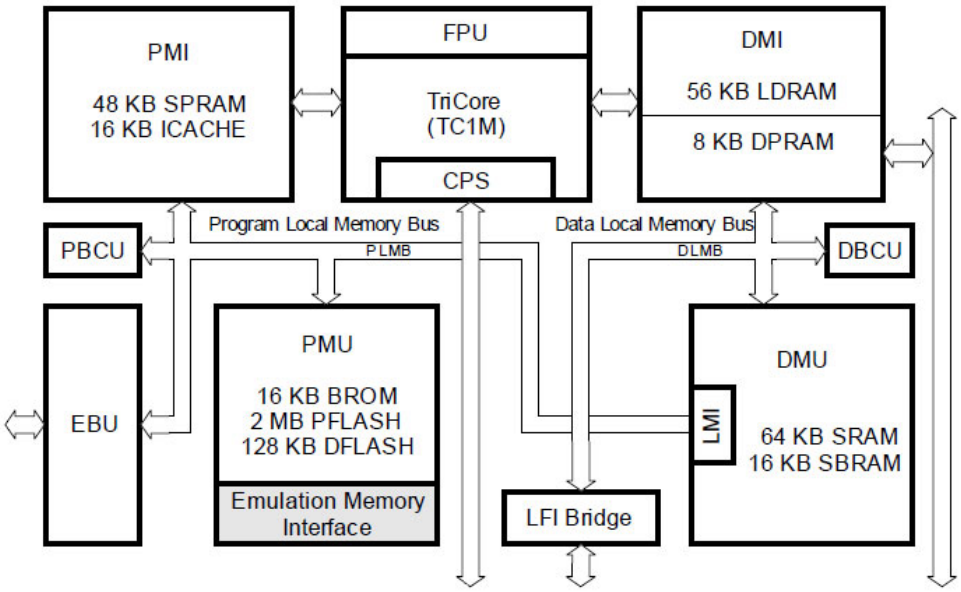


Abbildung 2.1.: Blockdiagramm des TC1796-Prozessors

Die DSP-Erweiterungen des Mikrocontrollers wurden für schnelle Bearbeitung komplexer Sensorsignale konzipiert und weisen typische Eigenschaften von DSP-Architekturen wie Multiply-Accumulate Unit[1], Integerarithmetik mit Sättigung, Packet Operations[2] und spezielle Speicheradressierungsmodi auf. Des Weiteren bietet die Architektur Optionen wie einen gemischten 32/16-Bit Befehlssatz und Hardwareunterstützung für schnelle Bearbeitung von Interrupts (mit durchschnittlicher Reaktionszeit von 200 ns bis zur Ausführung der ersten Instruktion in der Interrupt-Routine). Dabei kommt ein *Context Save Buffer* (Abk. *CSA*) zum Einsatz, der bei Unterbrechungen oder Funktionsaufrufen automatisch den oberen Kontext der Adress- und Datenregister sichert. Als oberer Kontext werden dabei die Register 8 bis 15 bezeichnet. Diese hardwareseitige Unterstützung erlaubt es, auf die Sicherung der einzelner Register mit dem MOV-Befehl zu verzichten, und so die Funktionsaufrufe wesentlich effizienter zu gestalten. In jedem Zyklus der Ausführung werden mit voller Auslastung der Pipelines durchschnittlich 1.3 Instruktionen abgearbeitet. Dabei werden bei einer Taktfrequenz von 150Mhz die meisten Befehle in einem Zyklus ausgeführt. Für Gleitkommabefehle steht eine eigene FPU zu Verfügung, damit können diese parallel zu anderen arithmetischen Instruktionen ausgeführt werden. Weitere Merkmale der TriCore-Architektur sind ein flexibles Bus Interface für einfachen Anschluss von externen Speichern, ein DMA-Controller, mehrere Mehrzwecktimer, Analog-to-Digital-Converter (Abk. *ADC*) und ein on-chip Scratchpad-Speicher, der zur Benutzer- oder Compiler-gesteuerten Spei-

[1]Damit können oft ausgeführte Befehlsfolgen, die das Produkt zweier Zahlen zu einer dritten addieren, in einer Instruktion ausgeführt werden.

[2]Erlauben das Setzen von einzelnen Bits in Registern, was z.B. in Netzwerkpacket-Headern zum Einsatz kommt.

cherung von Daten und Instruktionen für schnellen Zugriff benutzt werden kann.

Für weitere Beschreibung der TriCore-Architektur sei hier auf die Dokumentation von Infineon verwiesen ([Inf07]).

2.1.2. Die ARM-Plattform

Im Laufe dieser Diplomarbeit wurde das Compiler-Framework des WCC im Hinblick auf die Retargierbarkeit für unterschiedliche Architekturen erweitert. Als Beispiel für eine neue Zielarchitektur wurde der ARM7TDMI-Prozessor von ARM (Abk. für *Advanced RISC Mashines*) gewählt. Der ARM7TDMI [ARM04] wurde für Einsatz in mobilen Geräten und Systemen mit niedrigem Energieverbrauch konzipiert. Bis heute gehört dieser Prozessor zu den meistverkauften und meistbenutzten eingebetteten Prozessoren mit breitem Einsatz in der Unterhaltungselektronik wie z.B. in dem iPod von Apple oder Game Boy Advance von Nintendo, in Mobiltelefonen (Nokia) oder auch in dem Automotive-Bereich. Der ARM7TDMI-Prozessor stellt eine Implementierung der ARMv4T-Architektur dar, die sich durch einen simplen Aufbau zugunsten eines niedrigen Energie- und Platzverbrauchs auszeichnet. In der Abbildung 2.2 ist der Aufbau des ARM7TDMI-Prozessors in einem Blockdiagramm dargestellt.

Der Kern des Prozessors besteht aus einem 32-Bit RISC CPU, der als Load/Store Von Neumann-Architektur realisiert ist und somit einen gemeinsamen Speicher für Daten und Instruktionen vorsieht. Weitere Merkmale der ARMv4T-Architektur sind 16 uniforme 32-Bit Mehrzweckregister mit einem weiteren Statusregister und eine dreistufige Pipeline mit Fetch-, Decode- und Execute-Phasen. Alle Befehle besitzen eine feste Breite von 32 Bit (16 Bit in *Thumb*-Modus), dadurch wird das Dekodieren und Ausführen eines Befehls in der Pipeline auf Kosten von Codegröße erleichtert. Die meisten Befehle werden so in einem Zyklus abgearbeitet. Die Speicherzugriffe erfolgen immer an den Halbwortgrenzen ausgerichtet, somit kann zwar ein größerer Speicherbereich adressiert werden, aber es werden keine so genannte misaligned-Speicherzugriffe erlaubt. Es können nur LOAD-, STORE- und SWAP-Befehle auf dem Speicher zugreifen.

Um den simplen Entwurf der Architektur zu kompensieren wurden einige weitere Eigenschaften implementiert. So besitzt jede Instruktion ein 4-Bit großes Conditional Code-Feld, das eine bedingte Ausführung jeder Instruktion ermöglicht. Dieses Feld steuert den Zugriff auf ein Status-Register, der unter Anderem vier Flags (Overflow, Carry, Negative, Zero) enthält. Dieses Register kann optional von den meisten arithmetischen und logischen Instruktionen verändert werden. So wird hier das Ergebnis der Vergleichsoperationen abgespeichert oder der Overflow-Flag bei einem Überlauf gesetzt. Auch kann z.B. ein SUB-Befehl den Negative-Flag auf 1 sezten, wenn das Ergebnis der Substraktion negativ ist. Die Verwendung dieses Feldes verringert zwar die Anzahl der Bits, die für Speicheradressierung verwendet werden, allerdings ermöglicht die Ausnutzung dieses Mechanismus Instruktionen an vielen Stellen des Programms einzusparen. So kann das Inkrementieren oder Dekrementieren von Werten effizienter gestaltet werden. Auch können damit kurze `If`-Konstrukten einfacher realisiert werden, indem die sonst notwendigen Branch-Anweisungen eingespart werden. Der folgende C-Algorithmus, der zur Bestimmung des größten gemein-

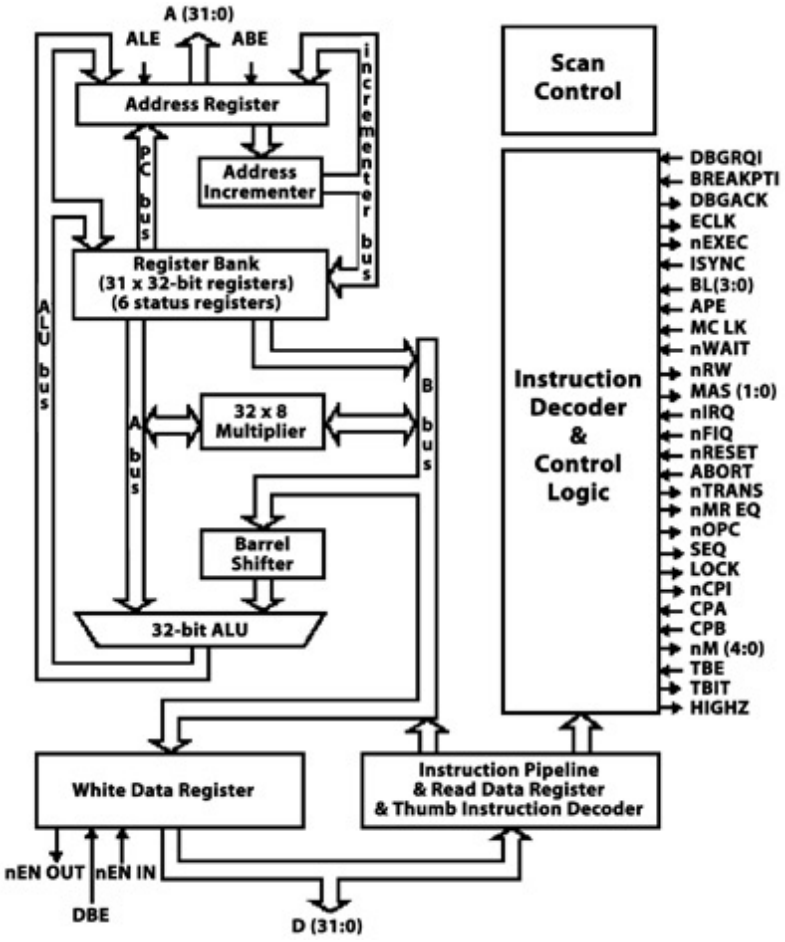


Abbildung 2.2.: Blockdiagramm des ARM7TDMI-Prozessors

samen Teilers eingesetzt wird,

```
while (i != j) {
  if (i > j)
   i -= j;
  else
   j -= i;
}
```

kann in ARM-Assembler wie folgt übersetzt werden:

```
loop CMP    Ri, Rj     ; Setze Condition Flag "NE" wenn (i != j)
                       ;                      "GT" wenn (i > j)
                       ;                  oder "LT" wenn (i < j)
     SUBGT  Ri, Ri, Rj ; wenn "GT" (greater than), i = i-j;
     SUBLT  Rj, Rj, Ri ; wenn "LT" (less than), j = j-i;
     BNE    loop       ; wenn "NE" (not equal), dann loop
```

Des weiteren gibt es einen 32-Bit Shifter, der gleichzeitig mit den meisten arithmetischen, logischen und Adressierungsbefehlen ohne Performanzeinbußen zum Verschieben oder Rotieren der Werte eines Registers verwendet werden kann. Damit wird kompakter Code mit wenigen Speicherzugriffen generiert, was sich wiederum in effizienteren Nutzung der Pipeline auswirkt. Weitere zusätzlichen Eigenschaften sind diverse Adressierungsmodi, sowohl direkt als auch über Basisregister, die das Laden und Speichern von einzelnen Halbwörtern und Wörtern erlauben. Die LDM- und STM-Befehle erlauben das Sichern und Auslesen von mehreren Registern gleichzeitig, was z.B. bei Funktionsaufrufen zur einfachen Rettung der Register benutzt wird. Die Register 8 bis 14 der 16 Mehrzweckregister sind so genannte Banked Register und werden abhängig von dem Operationsmodus des Prozessors (User, System, Interrupt etc.) automatisch durch andere, nur in jeweiligen Zustand verfügbare Register ersetzt. In allen Operationsmodi werden die Register 13 und 14 (*Stack Pointer* und *Link Register*) und zusätzlich im Fast Interrupt Mode (Abk. *FIQ*) die Register 8 bis 14 automatisch umgeschaltet. Das macht die explizite Rettung dieser Register beim Wechsel des Operationsmodus wie einem Interrupt überflüssig. Das Register 15 beinhaltet den *Program Counter* (PC). Außerdem wird für jeden Operationsmodus ein eigenes Status-Register benutzt, das somit ebenfalls nicht gerettet werden muss. Weitere Register können falls notwendig auf den Stack der aufrufenden Funktion gerettet werden. Für kurze Interrupts, die nur mit den Registern 8 bis 14 auskommen kann somit eine sehr schnelle Interrupt-Routine über FIQ realisiert werden, da kein Zeitverlust durch den sonst notwendigen Kontextwechsel auftritt.

Alle Prozessoren der ARMv4T-Architektur ab dem ARM7TDMI besitzen außerdem einen weiteren *Thumb*-Modus (das T in ARMv4T und in TDMI steht für Thumb). In diesem Modus führt der Prozessor einen 16-Bit Thumb Befehlssatz aus. Die meisten Thumb-Instruktionen werden dabei von entsprechenden ARM-Instruktionen durch Auslassen von einigen Operanden oder Begrenzung der Befehlsoptionen abgeleitet. Der so entstandene Befehlssatz hat weniger Funktionalität, so können z.B. nur Sprungbefehle bedingt ausgeführt werden und viele Thumb-Instruktionen können nur auf die Hälfte aller in der CPU verfügbaren Mehrzweckregister zugreifen. Der Vorteil von Thumb liegt in der meist höheren Codedichte, so dass für das gesamte Programm oft weniger Platz gebraucht wird. In Systemen, wo die Speicheranbindung über einen Bus mit weniger als 32-Bit Breite erfolgt kann die Verwendung von Thumb zu einer höheren Performanz führen, da weniger Programmcode in den Prozessor über den beschränkten Bus geladen werden muss.

Zu weiteren Details der ARMv4T-Architektur sei hier auf [ARM05] verwiesen.

2.2. Optimierende Compiler

Die Software für eingebettete Systeme muss bestimmte Kriterien erfüllen, die sie von Software für andere Architekturen unterscheidet, wie z.B. die Einhaltung bestimmter Echtzeitschranken oder Codegröße. Werden diese Kriterien nicht schon während der Programmkompilierung berücksichtigt, müssen die Programme später aufwendig getestet oder simuliert werden. Wenn die Kriterien nicht erfüllt werden, müsste die Software per Hand optimiert

und danach wieder den Tests unterzogen werden. Dieser Prozess kann sehr fehleranfällig und langwierig sein, was den Wunsch nach automatisierten Compilern nahe legt, die die Erfüllung solcher Kriterien schon während der Programmerstellung überprüfen können. Nach [Mar95] sollten die Compiler für eingebettete Systeme folgende Aspekte berücksichtigen:

1. *Schranken für die Echtzeitantwort des Programms.* Eingebettete Prozessoren müssen oft harte Echtzeitschranken einhalten, welche die Reaktionszeiten auf externe Ereignisse limitieren. Die Compiler müssen in der Lage sein, die Ausführungszeit des produzierten Codes zu berechnen und mit den angegebenen Zeitschranken zu vergleichen. Weiter entwickelte Compiler sollten mit Hilfe der Zeitschranken die Optimierungsvorgänge steuern können.

2. *Bedarf nach besonders schnellem Code.* Die Anforderung nach schnellerer Laufzeit des Programms wird oft durch den Einsatz von schnelleren Prozessoren beantwortet, um die Einhaltung fester Zeitschranken zu gewährleisten. Solche schnelleren Prozessoren sind in der Regel teuer und verbrauchen mehr Strom. Erhöhter Stromverbrauch ist oft unakzeptabel, wenn die Software auf tragbaren Geräten laufen soll. Damit hat die Aufgabe, schnellen Code zu produzieren größere Priorität als der Wunsch, kurze Kompilierzeiten einzuhalten. Deswegen können Compileralgorithmen, die in anderen Architekturen wegen erhöhter Laufzeitkomplexität abgelehnt werden für Compiler für eingebettete Systeme interessant sein.

3. *Bedarf nach kompakten Code.* Bei vielen Anwendungen wie z.B. Einchipsystemen (engl. *System on a Chip*, Abk. *SoC*) mit integriertem on-Chip Speicher ist der Platz zur Speicherung von Programmcode extrem limitiert. Für solche Systeme muss der Code sehr kompakt sein. Die Compiler müssen entsprechende Verfahren zur Reduzierung der Codegröße unterstützen, wie z.B. Verwendung von kürzeren Befehlssätzen (manche Prozessoren bieten neben 32-Bit auch 16-Bit Befehle an) oder Einsparung von Instruktionen (indem z.B. überflüssige NOP-Befehle vermieden oder nachfolgende ADD und MUL-Befehle durch einen einzigen Multiply-Add-Befehl ersetzt werden).

4. *Unterstützung für DSP-Algorithmen.* Viele eingebettete Systeme werden für digitale Signalverarbeitung (engl. *Digital Signal Processing*, Abk. *DSP*) benutzt. Entwicklungsplattformen müssen daher spezielle High-Level Unterstützung für diesen Anwendungsbereich anbieten. So soll es z.B. möglich sein, in einer Hochsprache Algorithmen mit Unterstützung von Fixpunktarithmetik, verzögerten Signalen, gesättigten arithmetischen Operatoren oder definierbarer Präzision für Zahlen zu spezifizieren.

5. *Unterstützung für DSP-Architekturen.* Viele eingebettete Prozessoren sind DSP-Prozessoren. Die Compiler müssen in der Lage sein, die besondere Architektur solcher Systeme zu benutzen. Dazu gehören Eigenschaften wie unterschiedliche Arten der vorhandenen Registersätze, parallele Ausführung von Befehlen (z.B. kann eine MAC-Instruktion (Abk. für *multiply and accumulate*) drei Zuweisungen gleichzeitig ausführen) oder spezielle Hardware für DSP-Algorithmen etc.

6. *Retargierbarkeit der Compiler.* Die Hardware der eingebetteten Systeme wird meistens in Abhängigkeit von der Anwendung ausgewählt. Um die schnelle Portierung der Software auf andere Plattformen zu ermöglichen müssen die Compiler in der Lage sein, Code für mehrere unterschiedliche Architekturen zu produzieren.

Die ersten drei Anforderungen an die Compiler können durch die Anwendung von Codeoptimierungen erfüllt werden. Diese analysieren den Programmcode und transformieren ihn auf die Weise, die zur Verbesserung des Programmverhaltens für das jeweilige Kriterium führt. Optimierungen innerhalb eines Compilers können dabei in drei unterschiedliche Gruppen aufgeteilt werden:

- *High-Level Optimierungen*: Diese Optimierungen sind unabhängig von der Zielarchitektur und werden auf einer Zwischendarstellung der Programmiersprache (sog. High-Level IR) durchgeführt. Einige Beispiele von solchen Optimierungen sind *loop tiling*, *loop blocking*, *loop fusion/fission*, *constant folding*, *constant propagation* usw.

- *Middle-Level Optimierungen*: Diese Optimierungen sind sowohl von der Hochsprache, in der das Programm geschrieben wurde als auch von der Zielarchitektur unabhängig. Als Zwischendarstellung kommt hier die Middle-Level IR zum Einsatz, die Eigenschaften wie 3-Adress-Code oder nur `goto`-Anweisungen zur Programmflusssteuerung aufweist. Damit lassen sich solche Optimierungen auf alle Quellsprachen und Architekturen anwenden. Beispiele für Optimierungen auf dieser Stufe sind *dead code elimination*, *partial redundancy elimination* oder *global value numbering.*

- *Low-Level Optimierungen*: Diese Optimierungen werden im Gegensatz zu den High-Level Optimierungen auf der maschinencodenahen Low-Level IR durchgeführt. Solche Optimierungen sind architekturspezifisch und nutzen die besonderen Möglichkeiten des Befehlssatzes und Aufbaus des jeweiligen Prozessors aus. Einige Beispiele dazu sind *correction of instruction types*, *avoidance of silicon bugs* oder *adjustment of jump displacements.*

Im Folgenden werden einige Compiler für eingebettete Systeme vorgestellt, welche den aktuellen Stand der Compilerentwicklung im diesem Bereich illustrieren. Im Abschnitt 2.2.1 wird die open-source Compiler-Suite *GCC* beschrieben, in 2.2.2 der energieoptimierende Compiler *encc* und in 2.2.3 schließlich der WCET-optimierende Compiler *WCC*. Wie sich zeigen wird, erfüllt keines von den hier vorgestellten Compiler-Frameworks die oben beschriebenen Anforderungen.

2.2.1. GCC

GCC [Sta08] ist der Name der Compiler-Suite des GNU-Projekts [gnu10]. Ursprünglich stand GCC für die Bezeichnung *GNU C Compiler*, da aber GCC außer C inzwischen auch einige weitere Programmiersprachen übersetzen kann, hat GCC heute die Bedeutung

GNU Compiler Collection (engl. für *GNU-Compilersammlung*) erhalten. Die Compiler-Suite enthält Compiler-Frontends für die Programmiersprachen C, C++, Java, Objective-C/C++, Fortran und Ada. Die Sammlung ist unter der GNU *General Public License* (Abk. *GPL*) lizenziert, die jedem Benutzer das Recht auf freie Verwendung, Änderung und Verteilung der Software garantiert [JKK+05].

Der GCC wird von einer Reihe von Betriebssystemen als Standardcompiler eingesetzt, darunter viele Linux-Distributionen, BSD, NextStep oder Mac OS X. Die Compiler-Suite wurde auf eine Vielzahl von unterschiedlichen Systemen und Hardware-Architekturen portiert, darunter auf eine Reihe von Prozessoren aus dem Bereich eingebetteter Systeme wie z.B. ARM, Infineon TriCore, Atmel AVR, Motorola 68000 und viele andere. Insgesamt kann mit GCC Code für mehr als 60 Plattformen erzeugt werden. Damit bietet sich GCC besonders für Betriebssysteme und Programme an, die auf vielen unterschiedlichen Hardware-Plattformen laufen sollen. Dies ist insbesondere für Software, die auf eingebetteten Systemen eingesetzt wird von Vorteil, da gerade hier schnelle Portierung und einfacher Wechsel der Zielarchitektur von großer Bedeutung sind.

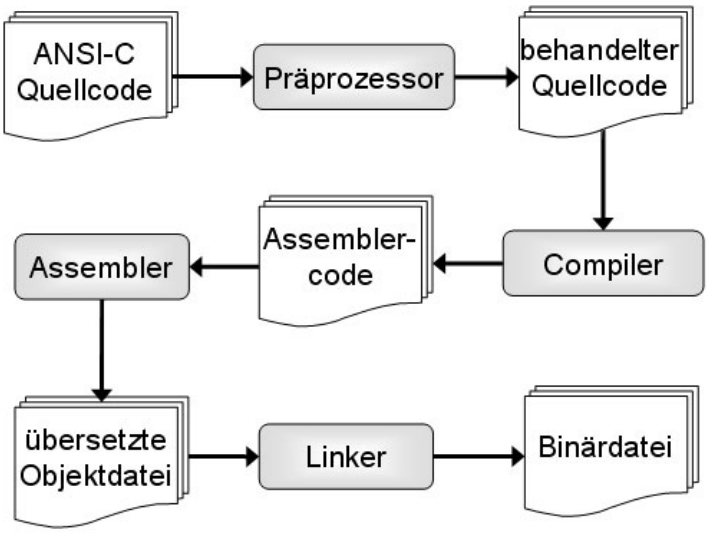


Abbildung 2.3.: Kompilierung eines C-Programms mit GCC

Die Abbildung 2.3 zeigt den Ablauf bei der Übersetzung eines Programms. Nach dem Aufruf des Hauptprogramms `gcc` wird die Sprache, in der der Quellcode verfasst ist ermittelt und abhängig davon die entsprechenden Kompilierprogramme aufgerufen. Für eine C-Quelldatei sind das Präprozessor *cpp*, Compiler *cc*, Assembler *as* und der Linker *ld*. Zuerst wird jede Datei von dem Präprozessor verarbeitet, indem alle Kommentare entfernt und die enthaltenen Präprozessormakros oder eingebundene Header-Dateien in reinen C-Code umgewandelt werden. Der Compiler parst den Code und transformiert ihn in eine Reihe von internen Zwischendarstellungen, auf denen verschiedene Codeoptimierungen durchgeführt werden. Schließlich wird aus der Zwischendarstellung der Assemblercode erzeugt. Dieser Assemblercode wird dann an den Assembler übergeben, der die Instruktions-Mnemonics in

binäre Maschinenbefehle übersetzt und als Ausgabe die Objektdateien erzeugt. Als Letztes wird der Linker aufgerufen, der alle Objektdateien eines Programms zusammen mit verwendeten Bibliotheken zu einer ausführbaren Binärdatei verbindet.

Da die eigentliche Optimierung eines Programms und Anpassung an die Zielarchitektur im Wesentlichen im Compiler *cc* stattfinden, soll an dieser Stelle dessen Aufbau noch etwas ausführlicher erläutert werden. Jeder GCC-Compiler besteht aus folgenden drei Komponenten: ein Frontend, ein Middleend und ein Backend. Der GCC verarbeitet immer nur eine Datei gleichzeitig, dabei durchläuft die Quelldatei alle drei Komponenten nacheinander. Die Abbildung 2.4 illustriert die einzelnen Komponenten und die Quellcode-Repräsentationen, die mit jeder Komponente assoziiert sind. Dabei entspricht die Aufteilung der Komponenten im Wesentlichen den am Anfang des Kapitel beschriebenen Code-Zwischendarstellungen. So operiert das Frontend auf einer High-Level IR, das Middleend auf einer Middle-Level IR und das Backend entsprechend auf einer Low-Level IR.

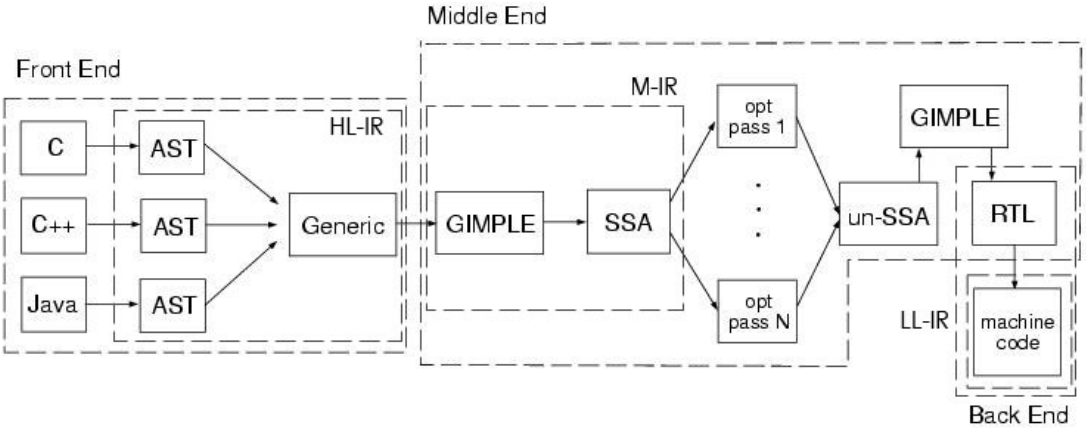


Abbildung 2.4.: GCC Frontend, Middleend und Backend

Die Aufgabe des Frontends besteht darin, den eingelesenen Quellcode in einen abstrakten Syntaxbaum (engl. *abstract syntax tree*, Abk. *AST*) zu transformieren. Für jede Programmiersprache gibt es ein eigenes Frontend. Da alle Sprachen verschieden gibt, ist auch das Format der für jede Sprache generierten ASTs z.T. sehr unterschiedlich. Um eine gemeinsame High-Level IR zu erhalten wird deswegen nach der Erzeugen der ASTs noch ein weiterer Schritt ausgeführt, in dem die Bäume in eine gemeinsame Form (sogen. *generic*) konvertiert werden. Eine Rücktransformation in den Quellcode ist ab dieser Stelle nicht mehr möglich, da alle quellsprachenspezifische Konstrukte notwendigerweise entfernt werden.

An dieser Stelle beginnt das Middlend des Compilers, das die High-Level IR in die Middle-Level IR konvertiert. Als erstes wird die komplexe AST-Darstellung in eine weitere Repräsentation mit dem Namen *GIMPLE* transformiert. In dieser Form besitzt jeder Ausdruck nicht mehr als drei Operanden, alle Kontrollflussverzweigungen sind als Kombinationen aus `goto`-Operatoren und bedingten Anweisungen dargestellt, Argumente einer Funktion können nur Variablen sein, etc. Auf der GIMPLE-Form kann somit eine Reihe

von sprach- und architekturunabhängiger Optimierungen durchgeführt werden. Dafür wird GIMPLE zusätzlich in die Static-Single-Assignment-Darstellung (Abk. *SSA*) transformiert. Sie zeichnet sich dadurch aus, dass jeder Variablen statisch nur einmal zugewiesen wird. Dadurch werden Datenabhängigkeiten zwischen Befehlen explizit dargestellt, was für viele Optimierungen von Vorteil ist. GCC kann mehr als 20 unterschiedliche Optimierungen auf den SSA-Bäumen anwenden, z.B. *Dead Code Elimination*, *Partial Redundancy Elimination*, *Global Value Numbering*, *Sparse Conditional Constant Propagation* oder *Scalar replacement of Aggregates*. Nach den SSA-Optimierungen werden die Bäume zurück in die GIMPLE-Form transformiert, welche danach zur Generierung von RTL-Darstellung (engl. *register-transfer language*, Abk. *RTL*) benutzt wird.

RTL ist eine hardwarebasierte Repräsentation, die auf einer verallgemeinerten Zielarchitektur mit einer unendlichen Anzahl von Register aufgebaut ist. Auf dieser Darstellung werden ebenfalls weitere Optimierungen wie *Common Subexpression Elimination*, *If-Conversion*, *Branch Probability Estimation*, *Sibling Valls*, *Constant Propagation* etc. durchgeführt. Zwar sind in der RTL-Darstellung weit weniger der für viele Optimierungen wichtigen High-Level-Informationen enthalten, dafür können an dieser Stelle maschinenabhängige Optimierungen angewendet werden, die Beschaffenheiten der Zielarchitektur ausnutzen. Es können z.B. vorhandene Instruktionen und deren Kosten oder der Aufbau der Pipeline miteinbezogen werden. Zuletzt generiert das Backend aus der RTL-Darstellung den Assemblercode für die Zielarchitektur. Auf dem Assemblercode kann der Assembler *as* noch weitere Optimierungen wie *Silicon Bugs Correction* durchführen.

Eine ausführliche Beschreibung der internen Funktionalität des GCC-Compilers kann auch in dem Artikel aus Red Hat magazine [Nov04] nachgelesen werden.

2.2.2. encc

Der *encc* (Abk. für *Energy Aware C Compiler*) [SWV10] wurde am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelt. Das Ziel bei der Entwicklung des *encc* war es, einen optimierenden Compiler zu erschaffen, der den Energieverbrauch des zu kompilierenden Programms berechnen und als Optimierungskriterium ausnutzen kann. Der encc-Compiler besteht aus folgenden Komponenten:

- Das Frontend LANCE2
- Ein Backend für den 16 Bit Thumb Befehlssatz des ARM7TDI-Prozessors (s. Kapitel 2.1.2)
- Ein Backend für den LEON 32 Bit Prozessor
- ARM Software Development Toolkit 2.50 (Assembler, Linker und Simulator)
- enProfiler, ein Tool zur Messung des Energieverbrauchs des Programms

Die Abbildung 2.5 illustriert den internen Aufbau und den daraus resultierenden Arbeitsablauf des encc. Der encc-Compiler verarbeitet ein in ANSI-C geschriebenes Programm

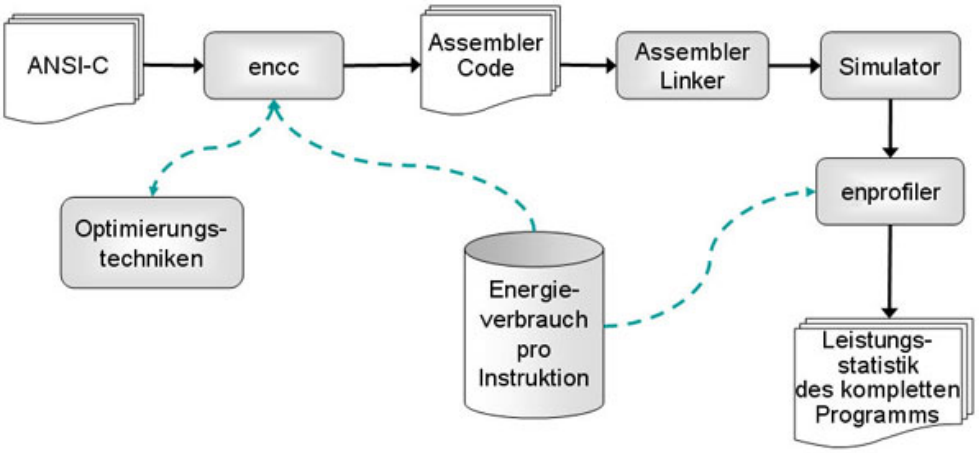


Abbildung 2.5.: Arbeitsablauf des encc

als Eingabe. Der Programmcode wird zuerst von dem *LANCE*-Frontend verarbeitet und im Hinblick auf den Energieverbrauch optimiert. Nach den anschließenden Phasen der Codegenerierung und Register-Allokation (im Bild nicht gezeigt) wird der Assemblercode des Programms generiert. Dieser Assemblercode wird vom Assembler und Linker zu einer Binärdatei kompiliert. Die Binärdatei wird auf einem Simulator für die Zielarchitektur ausgeführt. Der Simulator produziert als Ausgabe eine Trace-Datei des Programms, die Informationen wie Anzahl und Art der ausgeführten Instruktionen oder Zugriffe auf unterschiedliche Speicher beinhaltet. Diese Trace-Datei wird an das Analysetool *enprofiler* weitergeleitet, wo die Datei mit einem Parser eingelesen wird, um anschließend den Energieverbrauch der Instruktionen und den Speicherzugriffen zu berechnen. Die einzelnen Werte werden unter Beachtung der Zwischeninstruktionseffekte aufsummiert und zu einer Gesamtstatistik des kompletten Programms zusammengestellt. Diese Statistik enthält die Anzahl der ausgeführten Instruktionen, Anzahl der gebrauchten Prozessorzyklen, Anzahl und Art der unterschiedlicher Speicherzugriffe sowie schließlich den Energieverbrauch des Programms.

Das LANCE-Frontend wurde ebenfalls an der Technischen Universität Dortmund entwickelt und wird in der Forschung und für kommerzielle Compiler benutzt. Es repräsentiert den C-Quellcode in einer eigenen Zwischendarstellung, die auf einem Drei-Adress-Code arbeitet. Komplexe Programmkonstrukte wie Schleifen und If-Anweisungen werden durch `goto`-Befehle und Labels ersetzt. Somit werden viele High-Level Informationen, die z.B. Rückschlüsse auf die Iterationshäufigkeiten von Schleifen erlauben, bereits im ersten Schritt bei der Generierung der Zwischendarstellung entfernt. Ein Teil von LANCE2 ist eine Bibliothek, die einige gebräuchliche High-Level Optimierungen auf der Zwischendarstellung realisiert, wie z.B. *Jump Optimization*, *Common Subexpression Elimination* oder *Dead Code Elimination*.

Der encc besitzt keine Low-Level Zwischendarstellung, sondern erzeugt direkt einen Assemblercode, der vom ARM-Assembler eingelesen wird. Dazu verwendet das Backend

des encc-Compilers einen Tree-Pattern-Matching-Algorithmus, um die Codegenerierung durchzuführen. Für die Baumüberdeckung wird der Baum mit den geringsten Kosten ausgewählt, die Kosten spiegeln dabei den Energieverbrauch der Instruktionen wider. Es wird eine globale Register-Allokation durch Färben des Interferenz-Graphen durchgeführt. Für die Abschätzung der Spillingkosten wird für jede Schleife im Kontrollflussgraphen eine konstante Ausführungshäufigkeit angenommen. Dadurch werden Variablen, die sehr weit außen verwendet werden, zuerst gespillt. Nach der Register-Allokation und anschließender Instruktionsanordnung werden verschiedene Backend-Optimierungen angewandt, unter Anderem die compilergestützte statische Scratchpad-Speicher Allokation [SWLM02]. Dabei werden die Ausführungs- und Zugriffshäufigkeiten unabhängig von Eingabedaten festgelegt. Für jede Schleife wird vereinfacht eine zehnmalige Ausführung und für jede Programmflussverzweigung 50%-ge Wahrscheinlichkeit angenommen. Weitere Low-Level Optimierungen wie *Loop Transformation*, *Redundant Load/Store Elimination* und andere waren für die Zukunft geplant.

In den encc-Compiler ist auch eine Datenbank mit Informationen über den Energieverbrauch einzelner Instruktionen (erhalten durch physische Messungen an den ARM7TDMI-Prozessor) integriert, so dass alle Optimierungen mit Hinblick auf den gesamten Energieverbrauch des Programms evaluiert werden können. Diese Analyse wird Mithilfe des Analysetools enProfiler durchgeführt. Der enProfiler ist eng mit dem encc-Compiler verbunden und verwendet die gleiche Datenbank, um die Energiekosten der einzelnen Instruktionen zu bestimmen. Zusätzlich werden im enProfiler die Inter-Instruktions-Kosten, die bei einem Befehlswechsel entstehen, sowie die Speicherzugriffe mit in die Berechnung eingezogen. Die Betrachtung der Speicherzugriffe führt zur Notwendigkeit, das Speicherlayout des Zielsystems mit den dazugehörenden Datenbusbreiten, Wartezyklen, Caches und Energieverbrauch zu definieren. Diese Informationen wurden ebenfalls durch physische Messungen erhalten (für Caches wurden Angaben aus [WJ96] verwendet) und werden sowohl bei der Codegenerierung durch den encc-Compiler als auch bei der Evaluierung durch den enProfiler berücksichtigt.

Außer dem ARM7TDMI-Prozessors wurde der encc-Compiler auf den LEON-Prozessor retargiert. Dies ist eine SPARC V8-Implementierung, die in Form von VHDL-Quellcode verfügbar ist und unter der LGPL-Lizenz (Abk. für *Lesser General Public License*, einer neben der GPL weiteren von der Free Software Foundation entwickelten Lizenz für Freie Software) vertrieben wird. Mit diesem Prozessormodel ist es möglich, den LEON bis zum Gate-Level zu synthetisieren und die Schaltaktivität über Simulation zu bestimmen. Die Schaltaktivität ist ein Maß für den Energieverbrauch des Prozessors. Der Vorteil des frei verfügbaren VHDL-Models ist die Möglichkeit von Änderungen an der Hardware. Nach einer erneuten Synthetisierung können die neuen Energiewerte über Simulation ermittelt und mit den ursprünglichen Werten verglichen werden. Auf diesem Weg könnten die Optimierungen sowohl auf der Hardware wie auch auf der Software (über den encc-Compiler) evaluiert werden.

2.2.3. WCC

Die in dieser Diplomarbeit realisierten Compilererweiterungen wurden in das Compiler-Framework WCC (Abk. für *WCET-aware C Compiler*) [FLT06] integriert, das deshalb an dieser Stelle ausführlicher vorgestellt wird.

Der am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelter WCC stellt einen auf die WCET-Optimierung ausgelegten Compiler dar, der für die in ANSI-C geschriebene Programme Maschinencode für den Infineon TriCore TC1796 [Inf07] bzw. TC1797 [Inf09] generiert. Die Intention bei der Entwicklung des WCC war es, von Anfang an ein ausgereiftes, industriell eingesetztes WCET-Analysetool in den Compiler zu integrieren, um schon während der Programmübersetzung Zugriff auf WCET-Informationen zu erhalten, die für die Entwicklung von WCET-Optimierungen genutzt werden können. Dafür wurde das kommerzielle Produkt aiT [Abs10] der Firma AbsInt verwendet, welches eine statische WCET-Analyse eines übersetzten Programms ermöglicht.

Im Folgenden wird zuerst der Aufbau des WCC und seiner Komponenten kurz beschrieben, mit anschließender ausführlicher Darstellung der Bereiche, mit denen sich diese Diplomarbeit am stärksten befasst. Zum Abschluss des Kapitels sollen die besonderen Herausforderungen bei Erweiterungen des Frameworks dargestellt werden.

Aufbau des WCC

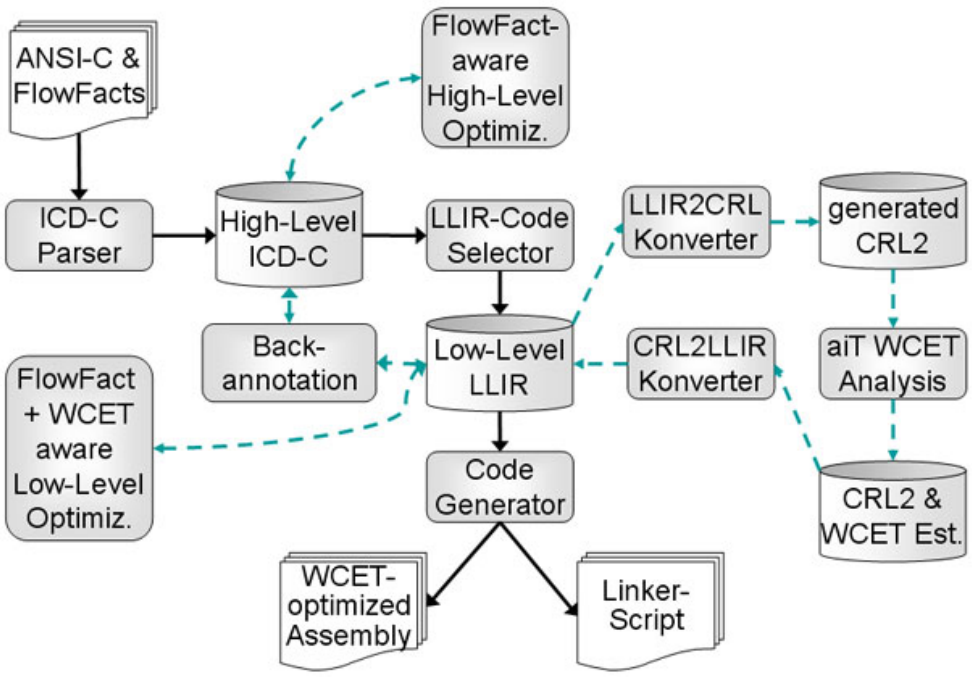


Abbildung 2.6.: Der Aufbau des WCET-aware C Compilers

In Abbildung 2.6 ist der Aufbau des WCC vor Beginn der Diplomarbeit zu sehen. Dabei wird durch die schwarzen Pfeile der Kontrollfluss innerhalb eines normalen Compilers angedeutet, gestrichelte Pfeile zeigen die Erweiterungen für WCET-Optimierung.

Als Eingabe liest der WCC in ANSI-C geschriebene Quelltexte eines zu übersetzendes Programms, das mit zusätzlichen Pragma-Direktiven (sog. *Flow Facts*) annotiert ist, die für die WCET-Analyse notwendige Kontrollflussinformationen wie die maximale Anzahl der Schleifeniterationen bereitstellen. Mit Hilfe des *Parsers* wird der Quelltext in eine interne Zwischendarstellung (engl. *Intermediate Representation*, Abk. *IR*) mit dem Namen *ICD-C IR* [PE10] übersetzt. Diese Zwischendarstellung ist eine direkte Repräsentation des C-Codes, und kann bei Bedarf wieder als Quelltext ausgegeben werden. Die ICD-C stellt eine Reihe plattformunabhängiger High-Level Optimierungen und Analysen zur Verfügung.

Mithilfe des eigens am Lehrstuhl entwickelten *Code Selectors* wird die High-Level IR in eine weitere Zwischendarstellung, die *ICD-LLIR* [EP10] (Abk. für *Low Level Intermediate Representation*), transformiert. Die LLIR ist eine maschinencodenahe Darstellung des Programms und besitzt ebenfalls diverse Analyse- und Optimierungsmöglichkeiten, sowohl plattformunabhängig als auch plattformspezifisch. Die Analysen werden zuerst auf der virtuellen LLIR durchgeführt. Diese besitzt eine unendliche Anzahl von Registern, in welchen die Programmvariablen abgelegt werden können. Durch eine weitere Optimierung die *Register-Allokation*, wird die virtuelle LLIR in eine physische LLIR übersetzt, welche nur die von der Zielarchitektur bereitgestellten Register belegt. Anschließend wird durch den *Codegenerator* die LLIR in Assembler-Dateien mit dem passenden Linkerskript überführt. Die Assembler-Dateien werden durch den TriCore-Assembler zu Object-Dateien kompiliert, aus denen danach mit dem Standard-Linker das binäre Programm erstellt wird.

Auf der LLIR-Ebene findet die schon früher erwähnte WCET-Analyse mit dem Analysetool aiT statt. Dazu wird im ersten Schritt die LLIR in das von aiT vorausgesetzte proprietäre Format CRL überführt. Diese Aufgabe wird von dem *LLIR2CRL-Konverter* übernommen, wobei unter Anderem der gesamte Kontrollflussgraph sowie die aus den C-Dateien extrahierten und während der Optimierungen konsistent gehaltenen Flow Facts ebenfalls in das CRL-Format übertragen werden. Die CRL-Datei wird von aiT analysiert und die Ergebnisse wieder in einer CRL-Datei abgelegt. Diese Informationen beinhalten im Wesentlichen die Ausführungshäufigkeiten und WCET der einzelnen Basisblöcke sowie die berechnete WCET für das gesamte Programm. Der *CRL2LLIR-Konverter* übertragt die Ergebnisse anschließend wieder in die LLIR, damit auf dessen Grundlage weitere Optimierungen ausgeführt werden können. Um WCET-minimierende High-Level Optimierungen zu ermöglichen, werden mit Hilfe der *Backannotation* die nur in der LLIR enthaltenen WCET-Informationen in die ICD-C transformiert.

Im Folgenden wird die ICD-LLIR genauer betrachtet, da sie die Codegrundlage fur die in dieser Diplomarbeit zu realisierenden Framework-Erweiterungen darstellt. In 2.2.3 soll die Transformation der ICD-C in die LLIR durch den Codeselector dargestellt werden und in 2.2.3 die Anbindung von aiT für die WCET-Analyse, die als Beispiel für die Anbindung von Profilern für weitere Optimierungskriterien dienen soll.

ICD-LLIR

Die ICD-LLIR ist eine vom Informatik Centrum Dortmund entwickelte maschinencodenahe Zwischendarstellung des Programmcodes. Sie wird innerhalb des WCC aus der ICD-C IR durch den Codeselector erzeugt. Im Gegensatz zu der hochsprachennahen Darstellung der ICD-C wird das Programm in der LLIR in der Form von Basisblöcken und architekturspezifischen Instruktionen dargestellt. Durch diese Realisierung lässt sich die Datenstruktur der LLIR sehr einfach in Assemblercode umwandeln.

Ein wesentliches Kriterium bei der Entwicklung der ICD-LLIR war die Retargierbarkeit, sodass die LLIR mit vertretbarem Aufwand auf unterschiedliche Architekturen und Prozessortypen (wie DSPs, VLIWs etc) angepasst werden kann. Aus diesem Grund ist der Aufbau der LLIR sehr allgemein gehalten, so dass für die Umstellung auf andere Architekturen nur die Prozessorbeschreibung ersetzt werden muss.

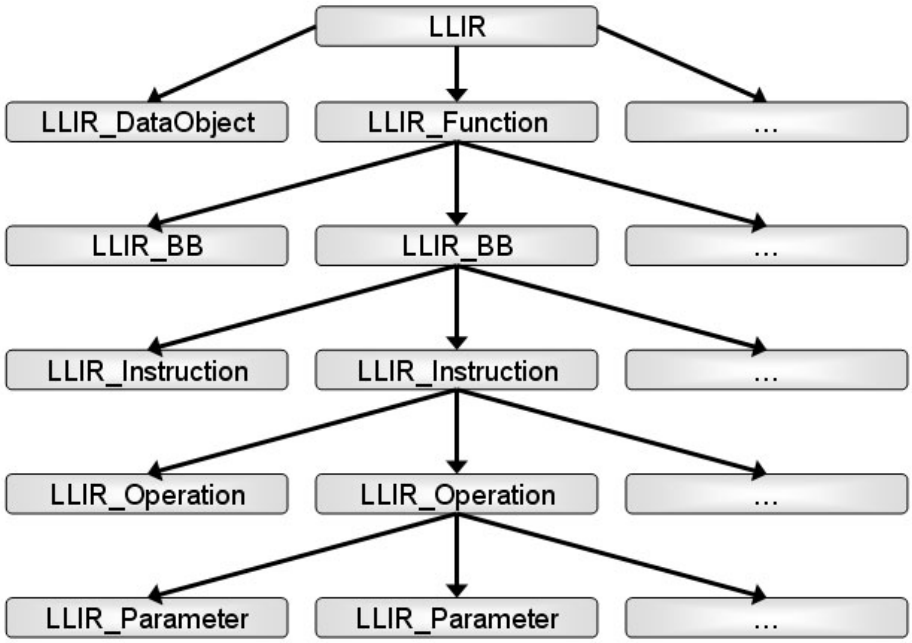


Abbildung 2.7.: Klassenhierarchie der ICD-LLIR

Die Abbildung 2.7 zeigt den genauen Aufbau der ICD-LLIR-Klassen. Die Hierarchie ist dabei wie folgt aufgeteilt:

- **LLIR** – Die Hauptklasse der LLIR. Entspricht jeweils einer Quelldatei des Eingabeprogramms. Beinhaltet die Funktionen und Datenobjekte der Klassen LLIR_Function bzw. LLIR_DataObject
- **LLIR_Function** – Eine LLIR_Funktion repräsentiert eine Assembler-Routine und besteht wiederum aus Basisblöcken der Klasse LLIR_BB.

- **LLIR_DataObject** – Ein LLIR_DataObject beinhaltet die Werte der Variablen eines Programms.

- **LLIR_BB** – Ein Basisblock ist eine Sequenz von Instruktionen, die nur einen Eintrittspunkt (d.h. nur die erste Instruktion kann das Ziel eines Sprungs sein), einen Ausgangspunkt und keine Sprunganweisungen innerhalb der Sequenz hat. Dabei kann das Ende des Basisblocks aus einem Sprungbefehl oder Funktionsaufruf oder Return bestehen. Der LLIR_BB besteht aus Instruktionen der Klasse LLIR_Instruction.

- **LLIR_Instruction** – Eine Instruktion ist eine Anweisung für den Zielprozessor. Bei den hier betrachteten Architekturen (TriCore und ARM) umfasst eine LLIR_Instruction genau eine Operation der Klasse LLIR_Operation. Im Falle eines VLIW-Prozessors (engl. *Very Long Instruction Word*) kann eine Instruktion aus mehreren Operationen bestehen.

- **LLIR_Operation** – Eine Operation ist ein einzelner Maschinenbefehl mit der dazugehörenden Assembler-Mnemonik und von der Operation benötigten Parametern, die durch die Klasse LLIR_Parameter repräsentiert werden.

- **LLIR_Parameter** – Ein LLIR_Parameter stellt einen Parameter in einer LLIR_Operation dar und kann hierbei eine Integer-Konstante, ein Register, ein Label oder ein Operator für den Speicherzugriff sein.

Die LLIR bietet diverse Analysen für den Daten- und Kontrollfluss eines Programms an. Eine wichtige Analyse, die auf der virtuellen LLIR ausgeführt wird, ist die sogenannte Lebendigkeitsanalyse (engl. *Life Time Analysis*, Abk. *LTA*). Hierbei wird die Lebenszeit der virtuellen Registern ausgewertet, die von der Register-Allokation benötigt wird, um zu bestimmen, auf welchen physischen Registern die virtuellen abgebildet werden können. Die Lebendigkeitsanalyse basiert im Wesentlichen auf einer weiteren Analyse, der DEF/USE-Analyse. Sie erlaubt die Aufstellung eines Abhängigkeitsgraphen für die Lese- und Schreibzugriffe auf jedes virtuelle Register, sodass für jede Instruktion und jedes Register bekannt ist, wann und durch welche Instruktion diese definiert oder benutzt werden und ob sie auch über Basisblockgrenzen hinweg erhalten bleiben müssen.

Desweiteren kann die LLIR um beliebige Benutzerdaten erweitert werden, ohne dass in die Struktur von LLIR selbst eingegriffen werden muss. Dazu werden die sogenannten Objectives benutzt. Diese werden von der Klasse LLIR_Objective abgeleitet und werden über die Klasse LLIR_Handler verwaltet. Damit kann ein beliebiges Objective-Objekt mit einer Instanz einer beliebigen Klasse in der LLIR-Hierarchie verknüpft werden, um sie mit weiteren Informationen zu versehen. Dabei wird jedes Objective-Objekt über den Typ (ObjectiveType) eindeutig identifiziert. Dieser Mechanismus wird im weiteren Verlauf dieser Arbeit dazu benutzt, um Ergebnisse der Analysen für weitere Optimierungskriterien wie ACET oder Energie in der LLIR verfügbar zu machen.

Codeselector

Der Codeselector stellt ein notwendiges und sehr wichtiges Bestandteil jedes Compilers dar, da an dieser Stelle die eigentliche Übersetzung der Quell- in die Zielsprache stattfindet. Der WCC-Codeselector transformiert die optimierte High Level ICD-C in die ICD-LLIR. Die gesamte ICD-LLIR Datenstruktur mit den dazugehörenden Objekten wie Funktionen, Basisblöcken, Operationen etc. werden an dieser Stelle erzeugt. Dabei wird für jede Quelldatei des Programms ein LLIR-Objekt generiert.

Die Transformation basiert auf der Methode des Tree-Pattern-Matchings, dabei wird für jeden Basisblock einer Funktion in der ICD-C ein Baum aufgestellt, der zur Auswahl von Assembler-Instruktionen für die Zielarchitektur genutzt wird. Die notwendigen Regeln zur Überdeckung des Baumes sowie eine Kostenfunktion zur Auswahl der günstigsten Überdeckung werden für den Code-Generator Generator IBURG/OLIVE [icd10] spezifiziert, der aus diesem Regelwerk dann den Codeselector produziert. Die Regelbeschreibung für TriCore ist sehr umfangreich, und umfasst mehr als 25.000 Zeilen Programmcode.

Im ersten Schritt der Transformation wird eine virtuelle LLIR erzeugt, der eine unbegrenzte Anzahl an virtuellen Register zu Verfügung steht. Das erleichtert die Anwendung von bestimmten Analysen und Optimierungen, da aber reale Prozessoren nur eine begrenzte Anzahl von Registern besitzen, muss in einem zweiten Schritt die virtuelle LLIR in die physische transformiert werden. Diese Transformation wird von der Register-Allokation übernommen, wie auf Seite 28 beschrieben.

Die Betrachtung des Codeselectors ist für die Diplomarbeit insbesondere deswegen von Bedeutung, da jede Architektur andere Instruktions- und Registersätze verwendet. Somit muss für eine Anpassung an weitere Architekturen auch der Codeselector an den jeweiligen Prozessor angepasst oder ersetzt werden.

WCET-Analyse

Wie schon oben erwähnt, wird für die Berechnung der WCET von zu optimierenden Programmen im WCC das statische Analysetool aiT eingesetzt, das auch von vielen namhaften Unternehmen aus der Industrie benutzt wird und sehr genaue statische Abschätzungen der oberen Schranken für Programmlaufzeiten liefert.

Die WCET-Analyse wird in mehreren Schritten unterteilt durchgeführt:

1. **Kontrollfluss-Analyse** rekonstruiert im EXEC2CRL-Dekoder den Kontrollfluss des zu untersuchenden Programms aus den ausführbaren Binärdateien.

2. **Value-Analyse** bestimmt die Schleifengrenzen, Inhalte der Register und berechnet die Adreßbereiche für Instruktionen, die auf den Speicher zugreifen.

3. **Cache-Analyse** nimmt eine Aufteilung der Speicherzugriffe in Cache-Hits und Cache-Misses auf.

4. **Pipeline-Analyse** berechnet das Verhalten der Prozessor-Pipeline unter Einbeziehung der Ergebnisse aus Schritt 2. und 3.

5. **Pfadanalyse** bestimmt den Worst-Case-Ausführungspfad für jeden Basisblock und für das gesamte Programm.

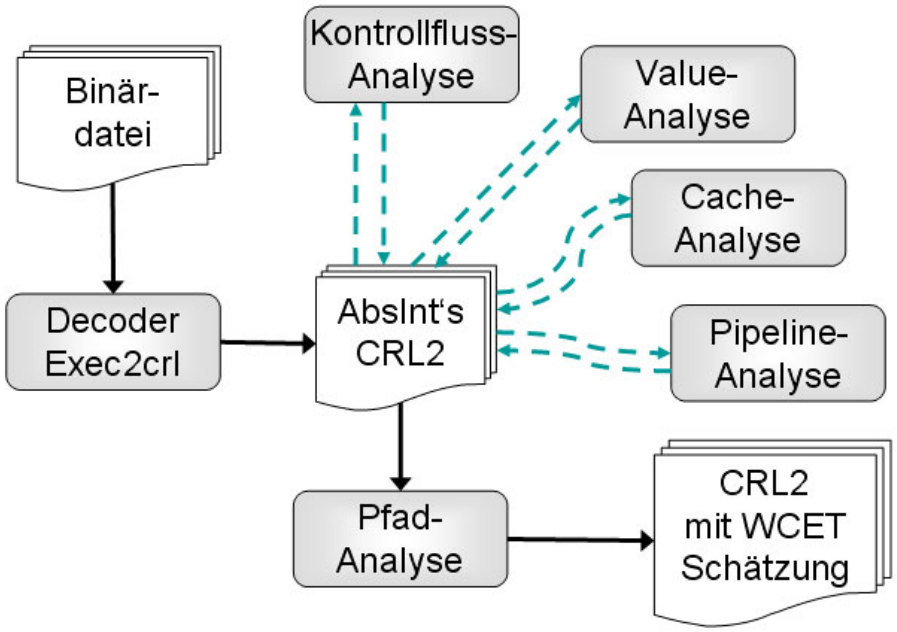


Abbildung 2.8.: Arbeitsschritte bei der WCET-Berechnung mit aiT

Die Reihenfolge bei der Abarbeitung der einzelnen Schritte ist in der Abbildung 2.8 dargestellt. Als Eingabe für das Analysetool dient eine ausführbare Binärdatei, die vom Compiler aus dem zu analysierenden Programmcode erzeugt wurde. Der Benutzer kann weitere Kontrollflussinformationen bereitstellen, wie z.B. Anzahl der Schleifendurchläufe oder Obergrenzen für Rekursionen. Im ersten Schritt der Analyse wird aus der Binärdatei mit Hilfe des EXEC2CRL-Dekoders der Kontrollfluss rekonstruiert und mit den bereitgestellten Zusatzinformationen wie der Anzahl der möglichen Schleifendurchläufe oder Obergrenzen für Rekursionen versehen, die für nachfolgende Analyseschritte notwendig sind. Dabei ist die Kenntnis über die Zielarchitektur von entscheidender Bedeutung, damit z.B. erkannt werden kann, welche Instruktionen Sprünge oder Funktionsaufrufe ausführen. Der so annotierte Kontrollfluss wird in das CRL-Format (engl. *Control Flow Representation Language*) übertragen, der die Grundlage für die weiteren Analysen darstellt. Das CRL-Format stellt eine maschinencodenahe Darstellung des Programms dar und ähnelt von seinem Aufbau her der LLIR. Aus diesem Grund ist die Konvertierung zwischen den beiden Formaten ohne Informationsverlust möglich. Die Transformation der LLIR in CRL mit dem LLIR2CRL-Konverter umgeht dabei den ersten Schritt im normalen Arbeitsablauf von aiT und macht die Verwendung des EXEC2CRL-Dekoders überflüssig. Der LLIR2CRL-Konverter erzeugt eine Zuordnung von LLIR-Basisblöcken zu den CRL-Basisblöcken, was

eine spätere Rückübertragung der Ergebnisse der WCET-Analyse in die LLIR ermöglicht.

Die anschließende Value-Analyse betrachtet die Werte in den Registern und Speichern, um nicht annotierte Schleifengrenzen und Adressen der Speicherzugriffe zu ermitteln. Diese Informationen sind für die nachfolgende Cache-Analyse von Bedeutung, damit für jeden Speicherzugriff bestimmt werden kann, ob Cache-Hit oder -Miss auftritt.

Bei der Pipeline-Analyse wird das Verhalten der Prozessor-Pipeline nachgebildet, mit dem Ziel, die genaue Ausführungszeit jeder Instruktion und jedes Basisblocks zu bestimmen. Alle Ergebnisse der vorangegangenen Analysen fließen hier ein und werden bei der Modellierung des aktuellen Zustands der Pipeline benutzt, wie z.B. die Belegung der Ausfühunrungseinheiten, Klassifizierung der Speicherzugriffe als Cache-Hits oder Misses, Timingsunterschiede bei Zugriffen auf verschiedene Speicher etc. Die hierbei ermittelten Ausführungszeiten der einzelner Instruktionen werden direkt in der Pfadanalyse verwendet, wo mithilfe der ganzzahligen linearen Programmierung aus dem Kontrollfluss eine sichere Vorhersage der WCET berechnet wird. Dabei wird für jeden Basisblock in der CRL-Datei seine Laufzeit und Anzahl der Aufrufe annotiert. Die Berechnung der WCET für das gesamte Programm ergibt sich aus der Summe der Laufzeiten der Basisblöcke, die auf dem längsten Pfad im Programm liegen.

Ein wichtiges Merkmal des Analysemodells von aiT ist dabei die Betrachtung der Kontexte von Schleifen und rekursiven Aufrufen. Die Cache-Kontexte von Schleifen oder Basisblöcken können sich stark unterscheiden, so erzeugt der erste Schleifenaufruf meistens einen Cache-Miss, wogegen bei weiteren Aufrufen die Instruktionen aus dem Cache geliefert werden können. Durch die Kenntnis dieser Kontexte lässt sich die Genauigkeit der WCET-Analyse also erheblich steigern.

Die auf diese Weise von aiT gewonnenen WCET-Informationen können nach der Rücktransformation durch den CRL2LLIR-Konverter dazu benutzt werden, um WCET-spezifische Optimierungen im WCC durchführen zu lassen.

Herausforderungen bei der Erweiterung des Frameworks

Auffällig bei der Betrachtung des Compiler-Frameworks ist die Tatsache, dass WCC keine völlig sprachen- und maschinenunabhängige Zwischendarstellung für Optimierungen wie MIR (engl. *Middle-Level Intermidiate Representation*) besitzt. Dadurch kann zwar eine höhere Qualität des erzeugten Codes erzielt werden, da prozessorspezifische Optimierungen angewendet werden können, andererseits wird so eine schnelle und einfache Portierbarkeit des Frameworks auf andere Architekturen erschwert. Der verwendete Codeselector unterstützt nur die TriCore-Architektur, müsste also für alle anderen Architekturen komplett ersetzt werden, was einen erheblichen Aufwand darstellt. Des weiteren existiert bisher kein Mechanismus, um weitere Optimierungskriterien außer der WCET zu betrachten. Durch die Erweiterung des Frameworks sollten sowohl die WCET-Informationen für weitere Architekturen nutzbar gemacht, als auch Grundlagen für die Optimierungen weiterer Kriterien geschaffen werden, indem weitere Profiler für zusätzliche Optimierungskriterien in das Framework integriert werden.

3. Retargierbares Compiler-Backend

Elektronische Systeme können grob in zwei große Bereiche eingeteilt werden: Mehrzweck- (engl. *general purpose systems*) oder Sonderzwecksysteme (engl. *special-purpose systems*). Mehrzwecksysteme wie PCs sind meistens auf Basis homogener Hardware aufgebaut und können vom Benutzer frei programmiert und konfiguriert werden, um den gestellten Anforderungen gerecht zu werden. Sonderzwecksysteme wie z.B. die eingebetteten Systeme sind dagegen oft dafür ausgelegt, nur eine einzelne Aufgabe in einem größeren Produkt zu bewältigen. Das führt zur einer großen Anzahl eingesetzter Systeme mit unterschiedlichen Architekturen und Befehlssätzen [Leu97]. Solche Systeme unterscheiden sich oft nur in der verwendeten Hardware und den Funktionen des übergeordneten Produkts. Dagegen sind einzelne Softwaremodule, die z.B. Signalverarbeitung übernehmen bei vielen Geräten gleich. Somit ist es aus Gründen der Zeit- und Kosteneffizienz wünschenswert, die auf diesen Systemen eingesetzte Software wiederzuverwenden. Des Weiteren existiert der Wunsch, die vorhandenen Entwicklungswerkzeuge auch für andere Architekturen zu verwenden. Dadurch verringert sich bzw. entfällt der Zeitaufwand für das Erlernen der Bedienung der Werkzeuge und die im Compiler vorhandene Optimierungen können auch auf weiteren Architekturen angewendet werden. Ebenfalls kann so die entwickelte Software schnell auf unterschiedlichen Architekturen getestet werden, um die passende Hardware für das geplante eingebettete System auszuwählen. Damit wird die Notwendigkeit von retargierbaren Compiler für eingebettete Systeme deutlich, welche die Übertragung und Entwicklung der Software auf anderen Architekturen erleichtern.

In diesem Kapitel werden Konzepte für die Erweiterungen an dem WCC-Framework vorgestellt, welche einen einfachen Austausch des verwendeten Compiler-Backends und somit der unterstützen Architektur ermöglichen. Damit wird die Grundlage für Evaluierung des Einflusses von Optimierungen zur Minimierung verschiedener Kriterien auf unterschiedlichen Architekturen geschaffen. Die entwickelten Erweiterungen basieren auf [PLM08], wo für solche Erweiterungen notwendigen Konzepte zuerst vorgestellt wurden. Als Beispiel wird ein retargierbares Compiler-Backend für die ARMv4T-Architektur entwickelt und beschrieben. Im Folgenden wird zuerst in Abschnitt 3.1 auf die retargierbaren Compiler im Allgemeinen eingegangen. In Abschnitt 3.2 werden dann die Erweiterungen am Framework beschrieben, die für eine erfolgreiche Integration einer neuen Plattform notwendig sind.

3.1. Grundlagen

Ein Compiler wird dann als *retargierbar* bezeichnet, wenn er für die Codeerzeugung für unterschiedliche Zielarchitekturen angepasst werden kann. Dabei soll der größte Teil vom

Quellcode des Compilers erhalten bleiben. Die Implementierung basiert auf einer formalen Beschreibung des Zielprozessors, die als zusätzliche Eingabe von dem Compiler eingelesen wird. Anhand dieses Prozessormodels wird der Compiler rekonfiguriert, um den Maschinencode für die aktuelle Zielarchitektur zu produzieren. Im Gegensatz dazu haben traditionelle Compiler ein fest eingebautes Modell des Zielprozessors, das für die Codeerzeugung eingesetzt wird.

Es wird unter unterschiedlichen Stufen der Retargierbarkeit unterschieden [Leu97]:

- *Prozessor-spezifisch*: Ein bestimmter Zielprozessor ist fest in den Compiler einkodiert, und die Techniken zur Codegenerierung unterstützen nur diesen einen Prozessor. Der größte Teil des Compiler-Quellcodes muss für die Unterstützung einer anderen Zielarchitektur umgeschrieben werden.

- *Portabel*: Der Compiler kann von einem Programmierer auf eine andere Zielarchitektur portiert werden, indem das Backend des Compilers ausgetauscht wird. Dieser Prozess erfordert gründliche Kenntnisse über den Compiler, die im Allgemeinen nur die Entwickler des Compilers besitzen.

- *Benutzer-retargierbar*: Der Compiler verwendet eine externe Beschreibung der Zielarchitektur, die in Compiler-spezifischen Beschreibungssprache verfasst ist. Ein erfahrener Benutzer kann den Compiler auf eine andere Zielarchitektur retargieren, indem nur wenige Veränderungen am Quellcode vorgenommen werden.

- *Prozessorunabhängig*: Der Compiler verwendet eine externe Beschreibung der Zielarchitektur, die in Compiler-spezifischen Beschreibungssprache verfasst ist. Alle notwendigen Informationen über den Befehlssatz der Zielarchitektur, die für die Codegenerierung benötigt werden, sollen automatisch aus der Beschreibung extrahiert werden. Es sind keine Modifikationen am Quellcode notwendig.

- *Parameterabhängig*: Der Compiler kann nur für eine enge Klasse der Prozessoren retargiert werden, die auf einer gemeinsamen Architektur aufbauen. Es wird eine externe Prozessorbeschreibung benutzt, die nur numerische Parameter wie Registerbreite, Wortlängen oder Anzahl der Recheneinheiten vorgibt. Es sind keine Modifikationen am Quellcode notwendig.

Viele der existierenden Compiler sind portabel. Eine Retargierung solcher Compipler ist nur mit einem großer Aufwand und mit tief gehenden Kenntnissen möglich. Damit sollten die retargierbaren Compiler nach Möglichkeit Benutzer-retargierbar sein. Der allgemeine Ansatz dabei besteht aus der Verwendung von Sprachen zur Prozessorbeschreibung gemeinsam mit weiteren benutzerdefinierten Komponenten für architekturspezifische Details, die nicht von Prozessorbeschreibung abgedeckt werden. Ein Beispiel für eien solchen Compiler ist der im Abschnitt 2.2.1 beschriebene GCC. Die Retargierung von GCC erfolgt über eine Datei mit Prozessorbeschreibung in einem speziellen Format. Weitere Besonderheiten des Zielprozessor können über Makros und Supportfunktionen beschrieben werden.

Die Klasse der Benutzer-retargierbaren Compiler enthält viele weitere Compiler-Frameworks. Dazu gehören kommerzielle Tools wie CoSy [cos10] oder Chess [che10] sowie vorrangig in der Forschung benutzte Compiler wie Record [Leu97]. Manche Frameworks generieren die Compiler aus einem Prozessormodel in einer Hardwarebeschreibungssprache (engl. *Hardware Description Language*, Abk. *HDL*) wie VHDL oder Mimola anstelle einer Compiler-spezifischen Prozessorbeschreibungssprache. Damit kann die aufwendige Entwicklung von Prozessormodellen in Compiler-spezifischen Beschreibungssprachen vermieden werden, da die vorhandenen HDL-Modelle übernommen werden können. Auf der anderen Seite kann das Extrahieren eines detaillierten Befehlssatzes aus einem Low-Level HDL Modell schwierig sein. Eine breite Übersicht über die retargierbaren Compiler wird in [LM01] gegeben.

Durch die Erweiterungen am WCC-Framework, die im nächsten Abschnitt vorgestellt werden, wird der Compiler zu einem Benutzer-retargierbaren System. Dies ermöglicht eine Retargierung an weitere Zielarchitekturen mit relativ wenig Aufwand.

3.2. Integration einer Plattform

Eines der Ziele bei Erweiterung des WCC-Frameworks war der Wunsch, den Compiler retargierbar zu machen. Die dabei entstandenen Erweiterungen sollen eine einfache Ausrichtung des Frameworks auf weitere neue Plattformen ermöglichen, indem das retargierbare Compiler-Backend jeweils an die entsprechende Architektur angepasst wird. Im Folgenden wird am Beispiel des ARM7TDMI-Prozessors die Realisierung eines solchen retargierbaren Compiler-Backends dargestellt. Dabei werden auch die Erweiterungen am Framework beschrieben, die eine allgemeine Retargierbarkeit des Frameworks erreichen.

Als erster Schritt findet die Ersetzung des TriCore-spezifischen Codeselectors sowie die Anpassung des Compiler-Backends ICD-LLIR an die neue Architektur statt. Der WCC-interne Codeselektor wurde aus einer sehr umfangreichen Prozessorbeschreibung für den TriCore manuell generiert (s. auch Abschnitt 2.2.3). Da dieser Vorgang für jede neue Architektur sehr viel Zeitaufwand bedeuten würde, wird stattdessen ein Compiler für das Zielsystem benutzt, der die Codegenerierung übernimmt. Für das verwendete Beispiel wird der TriCore-Codeselector durch den ARM-GCC-Compiler ersetzt, der Assemblercode für den ARM7TDMI generiert. Da AbsInt's aiT neben einer großen Anzahl von unterschiedlichen Prozessoren und Compiler auch den ARM7TDMI-Prozessor unterstützt, kann nach der Anpassung der LLIR und des LLIR2CRL-Konverters auch für die ARM-Architektur eine WCET-Analyse realisiert werden.

Abbildung 3.1 zeigt den neuen Aufbau des Frameworks nach dem Ersatz des internen Codeselectors durch einen Zielcompiler. Bevor das Quellprogramm an den Zielcompiler übergeben wird, wird es in die ICD-C IR transformiert, auf welcher dann die High-Level Analysen und Optimierungen laufen. Die ICD-C dient dabei als Source-to-Source-Optimierer. Auf diesem Weg werden die gesamten High-Level Analysen und Optimierungen im WCC für alle unterstützten Architekturen genutzt. Der optimierte Code wird schließlich an den Zielcompiler übergeben, welcher daraus Assemblercode für den ARM7TDMI-Prozessor ge-

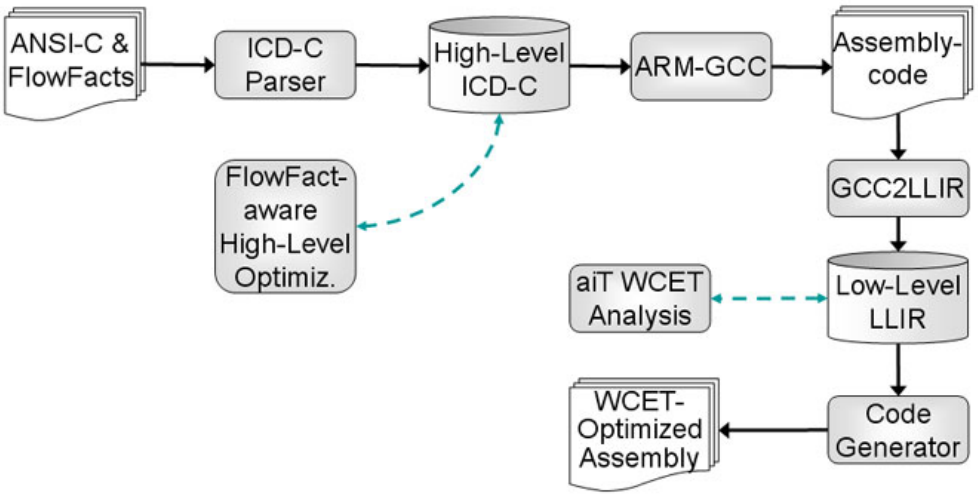


Abbildung 3.1.: Ersatz des internen Codeselectors durch einen Zielcompiler

neriert. Als Zielcompiler wurde der ARM-GCC (vergl. Kapitel 2.2.1) ausgewählt, der für die Übersetzung des vom WCC als Ausgabe produzierten optimierten C-Codes in Assembler-Befehle zuständig ist. Der GCC funktioniert an dieser Stelle wie eine Black Box, die nur die Transformation des Quelltextes in die Low-Level Assemblerdarstellung, sowie die anschließende Verlinkung der Objektdateien übernimmt. Als Optimierungsstufe wird `-O0` verwendet, d.h. es finden keinerlei Optimierungen im GCC statt. Es werden nur die WCC-eigenen Optimierungen auf der ICD-C IR und der LLIR benutzt. Dadurch kann sichergestellt werden, dass die Funktionen und Basisblöcke aus dem C-Quellcode unverändert in Assemblercode übertragen werden. Dies erlaubt eine einfache Zuordnung von Assembler-Routinen zu den entsprechenden C-Konstrukten, was vor allem die Rücktransformation der Ergebnisse der Analysen auf der Low-Level Ebene später ermöglicht.

In einem weiteren Schritt wird aus den mit dem GCC kompilierten Assemblerdateien die LLIR erzeugt. Diese Aufgabe wird von den GCC2LLIR-Parser übernommen. Die erzeugte LLIR kann danach mit dem für die ARM-Architektur angepassten LLIR2CRL-Converter als Eingabe für aiT dienen.

Im Folgenden wird in 3.2.1 die notwendige Anpassung der Prozessorbeschreibung in der LLIR beschrieben, 3.2.2 befasst sich mit der Integration des GCC als Codeselector in das Framework, und in 3.2.3 wird anschließend der GCC2LLIR-Parser erklärt.

3.2.1. LLIR

Um das Compiler-Backend an eine andere Zielarchitektur anzupassen, muss neben der Anpassung bzw. Ersetzung des Codeselectors auch die LLIR an die neue Architektur angepasst werden. Wie in 2.2.3 beschrieben, besitzt die LLIR einen generischen und einen architekturspezifischen Teil, sodass für die Umstellung auf eine andere Architektur nur die Prozessorbeschreibung ersetzt werden muss. Der genaue Vorgang ist in [EP07] beschrieben.

Im Wesentlichen sind dazu die Prozessorspezifikationsdateien *proc.h* und *proc.cc* zu ersetzen, zusätzlich müssen alle besonderen Konstrukte wie z.B. Offsets bei Adressierung über Label in der Parserspezifikationsdatei *llir3lex.lex* beschrieben werden. In diesen Dateien werden alle spezifischen Informationen über den Zielprozessor, insbesondere der Befehlssatz, Namen der vorhandenen Register und verwendete Pragmatypen mit Metainformationen für die LLIR definiert. Dabei lässt sich dieser Vorgang in die folgenden vier Schritte unterteilen:

- Modellierung der Registersätze
- Modellierung des Befehlssatzes
- Definition von Operatoren, die z.B. besondere Speicheradressierungsmodi beschreiben
- Definition von Pragmas

Zuerst müssen die verfügbaren physischen Registersätze der Zielarchitektur angegeben werden. Diese werden in *proc.h* als Präprozessor-Anweisungen mit `PHREG_` vor dem jeweiligen Registernamen definiert. Beispiele für die Register `R1`, `R2`, und den Program Counter:

```
#define PHREG_R1       "R1"
#define PHREG_R2       "R2"
#define PHREG_PC       "R15"
...
```

Alle 16 für den Programmierer sichtbaren physischen Register der ARMv4T-Architektur werden aus diese Weise hier definiert, zusammen mit den Namensaliassen wie `pc` für Register `R15` oder `lr` für `R14` etc. Ähnlich definiert werden die interne Hardwareressourcen, die nicht im Assemblercode abgebildet, sondern nur als implizite Operationsparameter benutzt werden. Bei ARMv4T sind das vor allem die von Operationen gelesenen und gesetzten Bits des Status-Registers. Diese werden mit `INTER_` vor dem Namen angegeben und können danach wie normale Register behandelt werden. Dabei kann der interne Status über die Funktion

```
bool LLIR_Register::IsInternal();
```

abgefragt werden. Auf diese Weise können auch die virtuellen Register über den Präfix `VREG_` definiert werden, um bei der Erzeugung der virtuellen LLIR auf sie zugreifen zu können. Der Code für das Compiler-Backend für die ARMv4T-Plattform kommt von dem GCC-Compiler, somit arbeitet das Backend nur auf der physischen LLIR. Aus diesem Grund werden hier auch keine virtuellen Register vorgegeben.

Der Befehlssatz der Zielarchitektur wird in der *proc.h* über eine enum-Aufzählung mit dem Typ `InstructionCode` definiert, und hat folgende Form:

```
enum InstructionCode
{
  INS_ADC_32=0,
  INS_ADD_32,
  INS_ADR_32,
  INS_AND_32,
  ...
}
```

Die Instruktionskosten (Größe in Bytes) werden über die Funktion `GetCost()` in der *proc.cc* berechnet und sind so in der gesamten LLIR-Hierarchie verfügbar. Die Byte-Kosten jeder einzelnen Instruktion werden dabei in einer Kostentabelle abgelegt. Da jede Instruktion in der ARMv4T-Architektur eine feste Wortbreite hat (32 Bit in ARM-Mode, 16 Bit in Thumb-Mode), ist eine statische Zuordnung in der Kostentabelle ausreichend. Des Weiteren können in diesem Schritt die sogenannten *Barriers* definiert werden, das sind Instruktionen, die im Code an einer festen Position stehen müssen. Solche Befehle werden über die Funktion

```
LLIR_Operation::createUnmovableStmt();
```

definiert und müssen danach von Optimierungen beachtet werden. So können z.B. alle Sprungbefehle als Barriers deklariert werden, um zu vermeiden, dass Scheduler-Algorithmen sie verschieben können. Ebenfalls in dieser zweiten Phase erfolgt die Zuordnung der impliziten Parameter zu den einzelnen Operationen, d.h. internen oder physischen Register, die von einer Operation benutzt, aber nicht im Assemblercode aufgelistet werden. Um sicherzustellen, dass alle diese Ressourcen trotzdem z.B. von der DEF/USE-Analyse berücksichtigt werden, werden die impliziten Parameter hier an die jeweiligen Operationen angehängt. Das kann statisch, z.B. über die Funktion

```
LLIR_Operation::createImplicitRegParam();
```

erfolgen, die zu jedem Objekt der entsprechenden Instruktion die impliziten Parameter zusammen mit der Art des Zugriffs (lesend, schreibend oder beides) hinzufügt. Eine weitere Möglichkeit ist die Definition einer Callback-Funktion, die bei jeder Änderung der Parameterliste einer Operation aufgerufen wird und dynamisch entscheidet, ob und welche impliziten Parameter benutzt werden. Schließlich muss für jede Instruktion und jede Operation (s. 2.2.3 für die Unterscheidung in der LLIR) eine Funktion definiert werden, welche die Erzeugung des entsprechendes Objektes mit den dazugehörenden Parameter in der LLIR übernimmt.

Im dritten Schritt der Anpassung der Prozessorbeschreibung können die sogenannten Operatoren definiert werden. Dies sind spezielle Parameter, die als Operationsparameter in dem Befehlssatz der Zielarchitektur verwendet werden können. Insbesondere können so die verschiedenen Adressierungsmodi bei Load/Store-Operationen modelliert werden. Die Operatoren werden ähnlich wie die Register in der *proc.h* über den Präfix `OPER_` definiert. Für die ARMv4T-Architektur wird so die Writeback-Option bei den LDM- und

STM-Operationen erfasst. Des Weiteren werden an dieser Stelle die Parameter für das Condition Code-Feld und den Shifter über enum-Aufzählungen definiert.

Im letzten Schritt werden die Pragmas, die zur Speicherung bestimmter Metainformationen dienen, wie z.B. die im Quellcode annotierten Flow Facts definiert. Dazu wird erstens in der enum-Aufzählung mit dem Typ `PragmaType` die unterschiedlichen Pragma-Typen aufgelistet. Dabei müssen an den ersten beiden Stellen die vordefinierten Pragmas `PRAGMATYPE_COMMENT` und `PRAGMATYPE_CUSTOM` stehen, wie in

```
enum PragmaType
{
  PRAGMATYPE_COMMENT=-1,
  PRAGMATYPE_CUSTOM,
  ...
}
```

Weitere Typen können beliebig vom Benutzer definiert und benutzt werden. Zu jedem in der Aufzählung eingetragenen Typ muss eine entsprechende textuelle Repräsentation geben, die als Präprozessor-Anweisung mit dem Präfix `PRAGMA_` definiert ist, z.B.:

```
#define PRAGMA_BYTE         "byte[ \t].*"
#define PRAGMA_SHORT        "short[ \t].*"
#define PRAGMA_WORD         "word[ \t].*"
...
```

Für die ARMv4T-Plattform wurden hier zusätzlich zu den vorhandenen Standard-Pragmas noch weitere von dem Codeselector GCC benutzte Pragmas hinzugefügt. Damit lassen sich die vom GNU-Assembler verwendete Direktiven an die entsprechende Funktionen und Basisblöcke anhängen.

3.2.2. GCC als Codeselector

Der GCC [Sta08] wird als Zielcompiler für den ARM7TDMI und als Ersatz für den internen Codeselector des WCC Compiler-Frameworks verwendet. Neben der Unterstützung für viele unterschiedliche Architekturen, für welche der GCC als Zielcompiler eingesetzt werden kann, war die freie Verfügbarkeit vom GCC einer der Gründe für diese Entscheidung. Des Weiteren wurden die Kommandozeilen-Schalter des WCC in Anlehnung an den GCC modelliert, was eine einfache Übertragung der Einstellungen ermöglicht. Vor allem kann so die gewünschte Optimierungsstufe über die Kommandozeilenschalter `-O0` bis `-O3` eingestellt werden. Die Integration des GCC in das Framework findet in der Datei *fsm.cc* der Klassenbibliothek *LIBFSM* statt. Über die Klasse `FSM` wird das WCC-Framework gesteuert, indem der gesamte Ablauf im Compiler über Zustände in einem endlichen Automaten (engl. *Finite State Machine*, Abk. *FSM*) definiert wird. Die enum-Aufzählung mit dem Typ `FSM_State` in *fsm.h* listet folgende mögliche Zustände auf:

- *INIT*: Initialisierungszustand des Compilers

- *MULTITASK_HANDLING*: Aufruf des Schedulers oder der multitask-Optimierungen
- *IR_GENERATION*: Generierung der ICD-C IR aus den Quelldateien durch den Parser
- *ICD_C_OPTIMIZATIONS*: ICD-C Optimierungen
- *CODESELECTION*: Aufruf des Codeselectors, der die ICD-C IR in die LLIR transformiert.
- *CODESELECTION_BUILTIN*: Interner Zustand wird zur Auswahl des eingebauten TriCore-Codeselectors
- *CODESELECTION_ARM_GCC*: Interner Zustand zur Auswahl des ARM-GCC als Codeselector
- *PROFILING*: Durchführung der ACET- und Energieanalyse
- *VIRT_LLIR_OPTIMIZATIONS*: Optimierungen auf der LLIR mit virtuellen Registern. Wird für ARM nicht angewendet.
- *INSTRUCTION_SCHEDULING_PRERA*: Scheduling von Instruktionen vor der Register-Allokation. Wird für ARM nicht angewendet.
- *REGISTER_ALLOCATION*: Aufruf der Register-Allokation. Wird für ARM nicht angewendet.
- *PHYS_LLIR_OPTIMIZATIONS*: Optimierungen auf der LLIR mit physischen Registern
- *INSTRUCTION_SCHEDULING_POSTRA*: Scheduling von Instruktionen nach der Register-Allokation
- *WCET_ANALYSIS*: Konvertierung von LLIR nach CRL2 mit WCET-Analyse, Rücktransformation und Annotierung in der LLIR.
- *BACKANNOTATION*: Annotation der WCET-Ergebnisse aus der LLIR zurück in die ICD-C IR
- *EMIT_ASSEMBLY_CODE*: Generierung von Assemblercode aus der LLIR
- *GENERATE_LINKER_SCRIPT*: Generierung des Linker-Skripts für das gesamte Programm, unter Beachtung des Speicher-Layouts, das vom WCC verwendet wird.
- *FINAL*: Letzter Zustand vor dem Verlassen des Automaten

Der FSM-Automat wird gestartet, indem ein Objekt der Klasse mit dem Start- und Endzustand als Initialisierungsparameter aufgerufen wird. Dabei können die Optimierungen auch einzelne Zustände aufrufen, um z.B. nur die WCET-Analyse durchzuführen. Der ARM-GCC wird ausgeführt, wenn der Compiler über die Kommandozeilenoption `-marm7` dazu aufgefordert wird, Code für die ARM-Plattform zu generieren. In diesem Fall wird bei dem Durchlauf der internen Zustände des Automaten der Codeselector auf ARM-GCC gesetzt und der Zustand `CODESELECTION_ARM_GCC` wird ausgeführt. Für jede durch den ICD-C IR optimierte Datei des Quellprogramms wird über einen System Call der ARM-GCC-Compiler aufgerufen, der aus den optimierten Quelldateien entsprechende Assemblerdateien generiert. Diese werden anschließend an den GCC2LLIR-Parser übergeben, um die LLIR zu erzeugen. Da die Register-Allokation an dieser Stelle durch den GCC bereits durchgeführt wurde, kann auf diesem Weg nur die physische LLIR erzeugt werden. Somit sind alle Optimierungen, die auf der virtuellen LLIR durchgeführt werden bei Benutzung von GCC als Codeselektor nicht anwendbar.

3.2.3. GCC2LLIR

Der GCC2LLIR-Parser hat die Aufgabe, aus den von ARM-GCC generierten Assemblerdateien die LLIR zu erzeugen. Der Parser ist als Standalone-Programm realisiert, welches als Eingabe die Assemblerdateien bekommt, und als Ausgabe die entsprechend transformierte LLIR im Form eines Datei-Dumps herausgibt. Die Dump-Datei kann danach in dem WCC mit der Funktion

```
bool LLIR::ReadFile(const char *name);
```

wieder eingelesen werden. Der GCC2LLIR-Parser wird über die Kommandozeile mit dem folgenden Befehl aufgerufen:

```
gcc2llir [-v] [ -o llirfile ] gccfile1 [ gccfile2 ... ]
```

Die Schalter in den eckigen Klammern stellen dabei optionale Parameter dar. Dabei wird mit `-o` der Name des LLIR-Dumps angegeben (ohne diesen Parameter wird der Dump standardmäßig in eine Datei mit dem Namen der Eingabedatei und der Endung **.llir* geschrieben), mit `-v` können Debug-Infos wie die gerade dekodierten Befehle angezeigt werden. Zuletzt werden die Namen einer oder mehreren Eingabedateien angegeben, die in die LLIR transformiert werden sollen.

Intern arbeitet das Programm als ein Textparser auf einer kontextfreien Grammatik, indem in einer while-Schleife jede Zeile eingelesen und analysiert wird. Die Funktionsweise des Parsers kann am folgenden Pseudocode veranschaulicht werden:

```
while (not end of file)
{
  read the line;
  if line == comment
      add new pragma_comment;
```

```
  else if line == global label && type == %function
      add new LLIR_Function;
  else if line == global label && type == %object
      add new LLIR_Dataobject;
  else if line == local label
      add new LLIR_BB;
  else if line == asm directive
      add new temp pragma_custom;
  else if line == mnemonic
   add new LLIR_Operation;
   check for parameter;
   add LLIR_Parameter to LLIR_Operation;
}
```

Das Format jeder Zeile muss den Konventionen des ARM-GNU-Assemblers [EF+07] entsprechen, so müssen z.B. alle Kommentare mit dem Zeichen `@` oder `#` anfangen, Label dürfen keine Leerzeichen am Anfang der Zeile besitzen, etc. Von besonderem Interesse sind hier die Assembler-Direktiven und die Instruktionszeilen. In jeder Zeile, wo eine Assembler-Direktive auftritt, wird geprüft, ob diese zu den so genannten Präfix-Direktiven gehört. Das sind Anweisungen, die vor einer Funktion oder einem Basisblock stehen und diverse Eigenschaften des nachfolgendes Blocks definieren, wie z.B. den Befehlssatz der Funktion (ARM oder Thumb), Größe der Funktion oder des Datenobjekts in Wörtern, die Speicher-Section, zu der die nachfolgenden Basisblöcke gehören usw. Gibt es in der LLIR keine dazu entsprechenden Parameter oder Operatoren, müssen diese Direktiven als Pragmas an das jeweilige Objekt angehängt werden. Dazu werden die Präfix-Direktiven in einer temporären Liste gespeichert, und an das nächste neue globale (Funktionen oder Datenobjekte) oder lokale (Basisblöcke) Objekt bei der Erstellung angehängt. Des Weiteren werden alle Labels aus der Assemblerdatei in eine LLIR-konforme Darstellung gebracht, indem z.B. alle führenden Punkte durch Unterstriche ersetzt werden.

Die Zeilen, welche die Instruktionen enthalten, werden nochmals in einer eigenen Schleife untersucht. Zuerst wird die Instruktion selbst über den Namen ermittelt, danach werden die Parameter der Instruktion gelesen und zu dem Instruktions-Objekt hinzugefügt. Dabei ist in einem zweidimensionalen Array `ptypematrixlong` für jede Instruktion abgelegt, welche Parameter an welcher Stelle stehen dürfen. Damit wird sichergestellt, dass alle eingelesenen Instruktionen auch tatsächlich dem in [EF+07] angegebenen Format entsprechen.

Schließlich wird für jeden erstellten Basisblock geprüft, ob dieser Sprunganweisungen enthält, die nicht als letzte Instruktion im Basisblock stehen. Werden solche Sprungbefehle gefunden, wird der Basisblock an dieser Stelle in zwei aufgeteilt, damit die Basisblockdefinition aus 2.2.3 nicht verletzt wird. Die Kontrollflusskanten für Nachfolger- und Vorgänger-Basisblöcke werden dabei entsprechend umgeleitet. Nachdem diese Phase erfolgreich abgeschlossen wurde, wird die LLIR über die Funktion

```
bool LLIR::WriteFile(const char *name, LLIR_WriteFileType mode);
```

in die Dump-Datei mit dem Namen `name` geschrieben, und in den WCC übernommen.

4. Multikriterielle Optimierungen

Moderne eingebettete Systeme müssen viele unterschiedliche, oft widersprüchliche Anforderungen gleichzeitig erfüllen. So erwartet der Benutzer von mobilen Geräten, dass diese möglichst günstig, klein und leicht sind, sehr lange Batterielaufzeit aufweisen und dennoch in der Lage sind, anspruchsvolle Anwendungen wie Videoverarbeitung in Echtzeit auszuführen.

Damit der Compiler die unterschiedlichen Anforderungen während der Codeerzeugung beachten kann, muss ihm die Kenntnis über die Auswirkungen von einzelnen Optimierungen auf mehrere unterschiedliche Kriterien vorliegen. Nur so kann der gewünschte Kompromiss zwischen Laufzeiteffizienz, Energieverbrauch und Codegröße erfolgreich gelingen. Da die Vorausberechnung der Auswirkungen mehrerer Optimierungen auf beliebigen Code unter Beachtung unterschiedlicher Kriterien unmöglich ist, wurde bei dieser Diplomarbeit ein anderer Weg gewählt. Dabei sollen die Auswirkungen der einzelnen Optimierungen auf das Programm direkt gemessen oder berechnet und den Optimierungen wieder zur Verfügung gestellt werden. Auf Basis dieser Informationen kann dann eine sogen. Feedback-Directed Optimierung [SS03] eingesetzt werden.

Ein Beispiel für ein solches Problem, das der Compiler bei multikriteriellen Optimierungen zu lösen hat, wird durch die Abbildung 4.1 aus einer aktuellen Forschungsarbeit am Lehrstuhl 12 veranschaulicht. In diesem Diagramm ist die Abhängigkeit zwischen zwei zu optimierenden Kriterien für ein Programm dargestellt. Die X-Achse beschreibt die Veränderung der relativen WCET in Prozent, und auf der Y-Achse wird die relative Codegröße (ebenfalls in Prozent) aufgetragen. Es ist leicht zu sehen, dass Optimierungen, die eine Minimierung der WCET erzielen, in diesem Fall eine Vergrößerung der Codegröße um bis zu 250% bewirken. Umgekehrt kann auch eine Verringerung der Codegröße nur über Erhöhung der relativen WCET erfolgen.

Im Folgenden werden in Abschnitt 4.1 die theoretischen Grundlagen vorgestellt, die bei Optimierungen von mehreren Kriterien beachtet werden müssen. Anschließend sollen die einzelnen in dieser Diplomarbeit betrachteten Kriterien näher vorgestellt werden. Insbesondere soll dabei auf die Messverfahren eingegangen werden, die für die Bestimmung der einzelnen Größen verwendet wurden. Die Berechnung der Laufzeit des Programms wird im Abschnitt 4.2.1 für die WCET und im Abschnitt 4.2.2 für ACET erläutert. Abschnitt 4.2.3 befasst sich mit der Berechnung des Energieverbrauchs eines Programms und abschließend behandelt Abschnitt 4.2.4 die Bestimmung der Programmgröße.

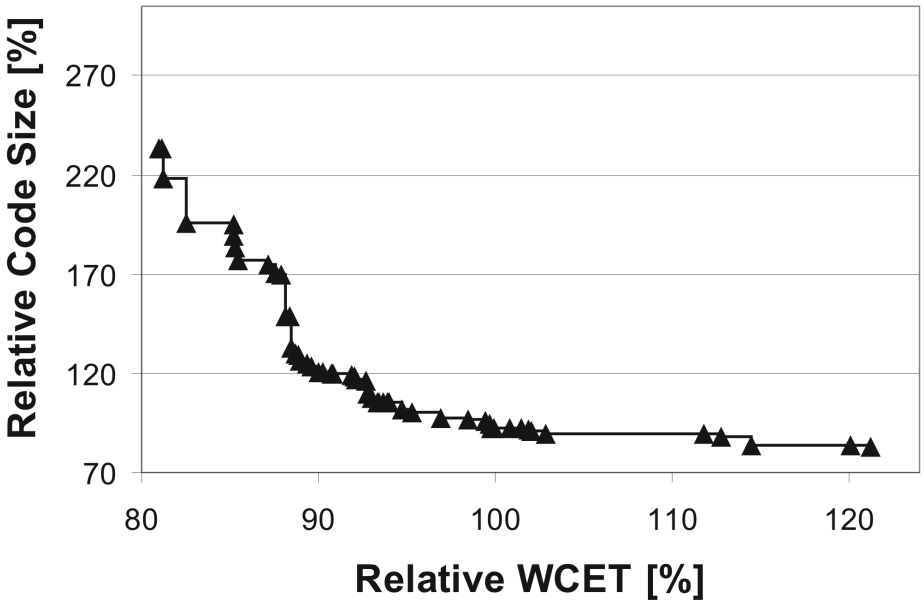


Abbildung 4.1.: Zusammenhang zwischen relativen WCET und Codegröße

4.1. Grundlagen

Eine Optimierung ist im Grunde ein Minimierungsproblem, da bei Compiler-Optimierungen immmer der Wunsch besteht, die entsprechende Zielgröße (oder auch Kriterium) wie maximale oder durchschnittliche Laufzeit, Energieverbrauch oder Codegröße zu verringern. Bezogen auf die Minimierung versucht somit eine multikriterielle Optimierung gleichzeitig m unterschiedliche Kriterien zu minimieren. Mathematisch lässt sich das wie folgt darstellen:

$$\text{Minimiere } y = f(x) = (f_1(x), f_2(x), \ldots, f_m(x))$$

wo x ein n-dimensionaler Entscheidungsvariablenvektor ist, $x = (x_1, \ldots, x_n) \in X$ und $y = (y_1, \ldots, y_n) \in Y$. X ist dabei der Raum aller möglichen Optimierungen und Y der Raum aller Kriterien. Jedes Kriterium hängt somit von dem Entscheidungsvektor x ab.

Ein oft angewendeter Ansatz zur Lösung solcher Probleme besteht darin, die zu optimierenden Kriterien als Teilkriterien aufzufassen und sie mittels Gewichtungsfaktoren zu einer gemeinsamen Zielfunktion zusammenzufassen. Auf diese Weise enthält man ein einfacheres Optimierungsproblem, das mit gängigen Mitteln wie z.B. ganzzahligen linearen Programmierung lösbar ist. Dieses Verfahren kann aber meistens nur bei relativ kleinen Problemen angewendet werden.

Wenn die Zielgrößen sich nicht einfach miteinander kombinieren lassen (so lässt sich der Energieverbrauch z.B. nur schwer in Gewicht darstellen), ist die Wahl der eingesetzten Gewichtungsfaktoren eher willkürlich und subjektiv. Dadurch ergibt sich ebenfalls eine entsprechende Willkürlichkeit bei der Bestimmung der gesuchten „besten“ Lösung des Op-

timierungsproblems. Ein alternativer Ansatz ist in solchen Fällen die separate Optimierung für alle möglichen Kombinationen von Gewichtungsfaktoren. In der Regel lässt sich dabei keine einzelne beste Lösung finden, die für alle Kriterien gilt, da die Zielkriterien wie oben beschrieben oft miteinander in Konflikt stehen. Stehen die Kriterien hingegen nicht in Konflikt, beschränkt sich das Problem darauf, alle Kriterien für sich allein zu optimieren, also die Lösung von m einkriteriellen Problemen zu finden.

Wenn keine eindeutig beste Lösung auffindbar ist, wird eine Menge von Lösungen des Optimierungsproblems bestimmt. Diese Menge ist dadurch definiert, dass eine Verbesserung eines Zielfunktionswertes nur durch Verschlechterung eines anderen erreicht werden kann. Diese Menge dominiert also alle anderen Lösungen. Ein Entscheidungsvektor $x \in X$ dominiert einen anderen Entscheidungsvektor $z \in X$ (geschrieben $x \prec z$) genau dann, wenn für alle Teilfunktionen die Lösung durch den Entscheidungsvektor x nicht schlechter als durch den Entscheidungsvektor z ist, und für mindestens eine Teilfunktion besser:

$$\forall i \in \{1, \ldots, m\} : f_i(x) \leq f_i(z) \quad \text{und} \quad \exists j \in \{1, \ldots, m\} : f_j(x) < f_j(z)$$

x dominiert z schwach (geschrieben $x \preceq z$) genau dann, wenn

$$\forall i \in \{1, \ldots, m\} : f_i(u) \leq f_i(v)$$

D.h. für alle Teilfunktionen ist die Lösung durch den Entscheidungsvektor x nicht schlechter als durch den Entscheidungsvektor z.
x dominiert z streng genau dann, wenn

$$\forall i \in \{1, \ldots, m\} : f_i(u) < f_i(v)$$

Streng dominierende Lösungen durch den Entscheidungsvektor x sind somit für alle Teilfunkionen besser, als durch den Entscheidungsvektor z. Ein Entscheidungsvektor $x \in X$ wird genau dann *Pareto-optimal* bezüglich X bezeichnet, wenn es keinen anderen Entscheidungsvektor in X gibt, der x dominiert.

Die Menge aller Pareto-optimalen Lösungen im Entscheidungsvariablenraum bildet dabei das *Pareto-Optimum*. Die entsprechende Menge der Vektoren aus dem Kriterienraum wird auch *Pareto-optimale Front* genannt. Somit wird bei multikriteriellen Optimierungen nach der entsprechenden Pareto-optimalen Front der Lösungen gesucht. Aus dieser Menge wird dann derjenige Punkt ausgewählt, der der gewünschten Optimierung an ehesten entspricht. Wenn z.B. in Beispiel 4.1 die Minimierung der WCET Vorrang hat, so kann einer der Punkte aus der Pareto-Front genommen werden, der auf der x-Achse ganz links steht. Die Abbildung 4.2 zeigt einen Beispiel für eine solche Pareto-optimale Front für die Kriterien f_1 und f_2. Die Front wird durch die dunkle Linie dargestellt. Die Punkte stellen die Ergebnisse nach der Anwendung unterschiedlicher Optimierungen dar. Für jede Lösung aus dieser Front gilt, dass eine Verbesserung des Wertes für das Kriterium f_1 (Punkte weiter links und unten auf dem Diagramm) nur dadurch erreicht werden kann, dass die Lösung für das Kriterium f_2 schlechter wird, und umgekehrt.

Für die Bestimmung von Pareto-optimalen Lösungen und Erzeugung der Pareto-optimalen Front existiert eine Vielzahl von unterschiedlichen Ansätzen, meistens aus der Klasse

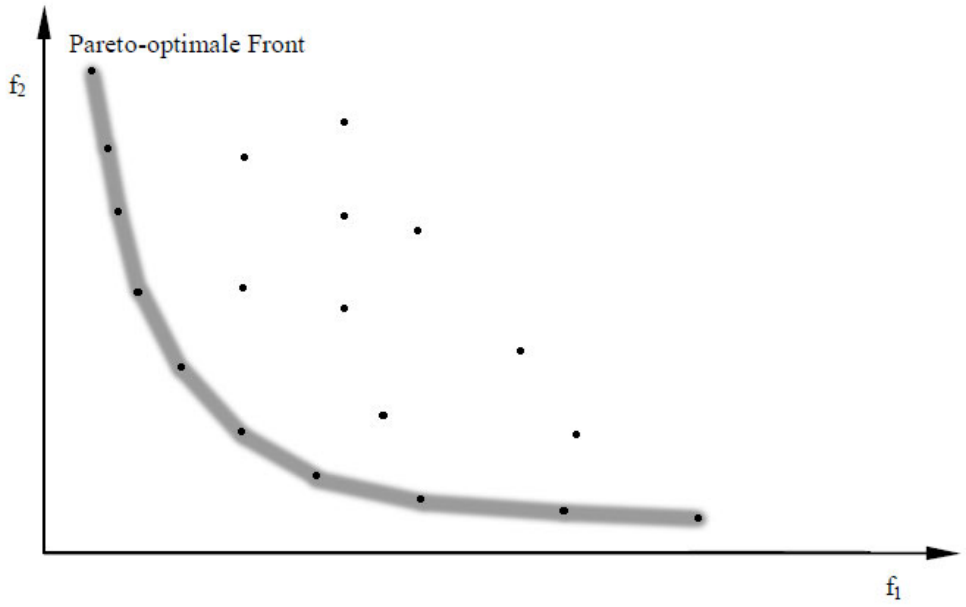


Abbildung 4.2.: Pareto-optimale Front für die Kriterien f_1 und f_2

der numerischen oder evolutionären Algorithmen. Die Hauptidee bei allen Ansätzen besteht dabei darin, eine Anfangsmenge zu bestimmen, die sich danach in mehreren Durchläufen mehr und mehr der Pareto-optimalen Menge nähert. An dieser Stelle soll ein kurzer Überblick über die bekanntesten Methoden gegeben werden.

4.1.1. Bestimmung von schwach-dominierenden Lösungen

Zuerst wird ein Verfahren vorgestellt, das bei einer gegebenen Menge P von Lösungen die schwach dominierende Menge P' bestimmt. Dieser Algorithmus geht alle möglichen Lösungen durch und vergleicht sie miteinander, um die schwach dominierende Menge zu erhalten. N bezeichnet nachfolgend die Größe von P und M die Größe des Zielfunktionsvektors.

Naiver Algorithmus

1. Setze Zähler i=0 und erzeuge schwach dominierende Menge $P' = \{\}$.

2. Für eine Lösung $j \in P \setminus i$, überprüfe ob Lösung $j \preceq i$. Falls ja gehe zu Schritt 4.

3. Wenn noch unbehandelte Lösungen in P existieren, $j = j + 1$ und gehe zu Schritt 2. Sonst $P' = P' \cup i$

4. $i = i+1$, falls $i \leq N$ gehe zu Schritt 2, sonst STOPP. P' ist die schwach dominierende Menge.

Die Laufzeit von diesem Algorithmus ist $O(N^2 \cdot M) \cdot (O(N) + O(M \cdot N)$ [Deb01]. Diese Vorgehensweise ist sehr langsam, da sie jede vorhandene Lösung mit allen anderen vergleichen muss, was sehr viele Durchläufe durch die Menge P notwendig macht.

4.1.2. Klassische Methoden

Es gibt eine Vielzahl von unterschiedlichen numerischen Methoden, die das Bestimmen von Pareto-optimalen Mengen behandeln. Hier sollen zwei sehr bekannte Methoden und ihre Nachteile vorgestellt werden.

Weighted Sum Ansatz

Dieser Ansatz besteht darin, den einzelnen Teilfunktionen mit Gewichtungsfaktoren zu versehen, um sie anschließend in einer gemeinsamen Zielfunktion zusammenzufassen. Diese kann danach wie ein einkriterielles Problem gelöst werden. Die Funktion hat die Form

$$\min \sum_{i=1}^{M} w_i \cdot f_i(x) \quad \text{u.d.N.}^1 \quad x \in X, w_i \geq 0, \forall i = 1, \ldots, M \quad \text{und} \quad \sum_{i=1}^{M} w_i = 1$$

Die Abbildung 4.3 veranschaulicht die unterschiedlichen Geraden, die für verschiedene Gewichte im 2-dimensionalen Fall entstehen. Jede Gerade a bis d ist die Realisierung der Formel $F(x) = w_1 \cdot f_1(x) + w_2 \cdot f_2(x)$. Durch die Wahl von verschiedenen w_i kann die Gerade im Lösungsraum verschoben werden. Die Minimierung bedeutet hierbei, dass ein Punkt an der Pareto-optimalen Front gesucht wird. Die minimale Lösung in diesem Fall wird durch die Gerade d realisiert, die die Pareto-optimalen Front in Punkt A schneidet.

Das Ergebnis hängt somit entscheidend von den zugeordneten Gewichtungsfaktoren. Des Weiteren kann dieser Ansatz nur Lösungen auf der konvexen Seite der Pareto-optimierten Front finden, alle anderen Lösungen werden nicht gefunden. Pro einen Lauf des Algorithmus kann jeweils nur eine Lösung aus der Front approximiert werden, für weitere Punkte muss der Algorithmus entsprechend mehrmals ausgeführt werden.

ϵ - Constraint Methode

Bei dieser Methode wird die ursprüngliche Zielfunktion verändert, indem nur das Minimum einer Teilfunktion gesucht wird (z.B. einer, die für das Problem am wichtigsten ist). Die restlichen Teilfunktionen werden in Nebenbedingungen mit der oberen Schranken ϵ transformiert. Somit hat die Funktion die Form

$$\min f_l(x) \quad \text{u.d.N.} \quad f_i(x) \leq \epsilon_j \quad \forall i = 1, \ldots, M, i \neq l, x \in X$$

Die Abbildung 4.4 zeigt den eingeschränkten Bereich durch die wählbaren ϵ im 2-dimensionalen Fall. Die Funktion f_1 wird durch die ϵ nach oben beschränkt, und bildet einen

[1] unter der Nebenbedingung

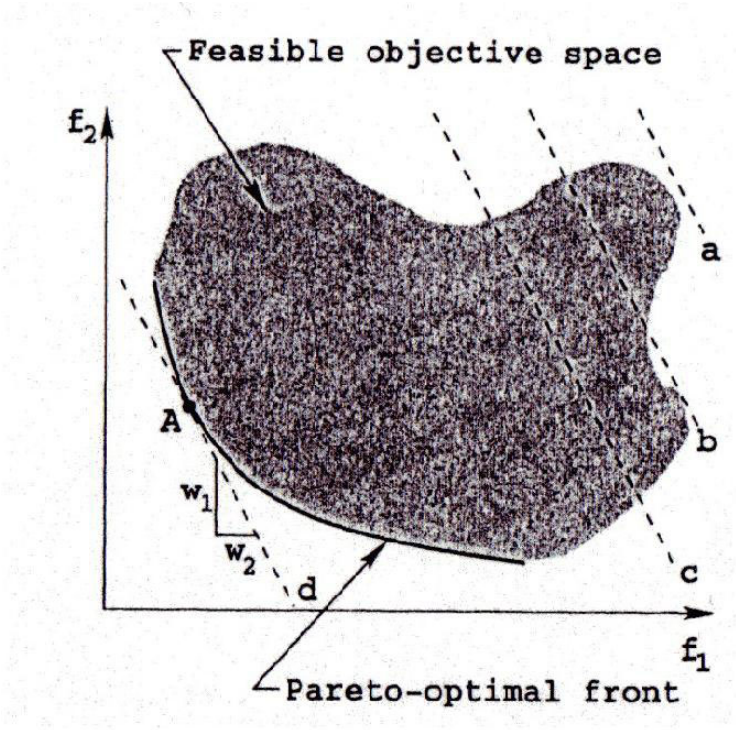


Abbildung 4.3.: Weighted Sum [Bur05]

Intervall $[0, \epsilon]$. Somit muss nur der minimale Punkt der Funktion f_2 in diesem Intervall gefunden werden. Für ϵ_1^b z.B. ist dies der Punkt B, und für ϵ_1^c der Punkt C.

Nach [Ehr00] ist die Lösung, die mit Festlegung des ϵ gefunden wird auch Pareto-optimal. Das Verfahren findet sowohl bei konvexen als auch bei nicht-konvexen Problemen eine Vielzahl von Pareto-optimalen Lösungen. Allerdings besteht die Schwierigkeit bei einer geeigneten Wahl von ϵ, da bei einer schlechten Wahl der Lösungsraum auch leer sein kann.

Dieser Ansatz eignet sich besonders für mehrkriterielle Optimierungen, wo eines der Kriterien die WCET ist. Die WCET stellt eine Schranke für die maximale Antwortzeit des Systems dar. Im Allgemeinen ist nur die Einhaltung dieser Schranke interessant, nicht aber die Effizienz, mit der dies geschieht. So beträgt z.B. bei einem Videoplayer, der 25 Frames pro Sekunde anzeigt, die Mindestantwortzeit 1/25 Sekunde. In dieser Zeit muss jeweils ein Frame berechnet werden. Eine doppelt so schnelle Ausführung ist nicht sinnvoll, da der nächste Frame nicht früher angezeigt werden darf. Dem gegenüber erwirkt z.B. eine Verringerung des Energieverbrauchs immer eine höhere Batterielaufzeit, und erhöht somit die Gesamteffizienz des Systems. Somit reicht es aus, für WCET-Kriterien nur für die Mindestantwortzeit zu optimieren. In diesem Fall kann die WCET-Schranke direkt als ϵ übernommen werden.

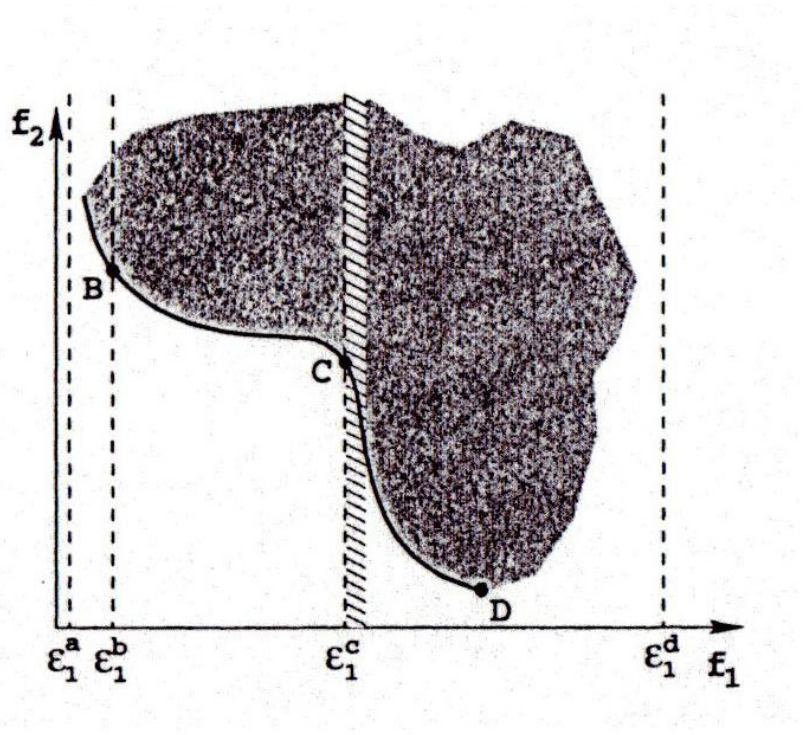


Abbildung 4.4.: ϵ - Constraint [Bur05]

4.1.3. Evolutionäre Algorithmen

Neuerdings findet die Berechnung der multikriteriellen Optimierungen mit evolutionären Algorithmen eine große Verbreitung. Ein evolutionärer Algorithmus (Abk. *EA*) ist ein Optimierungsverfahren, das am Vorbild der biologischen Evolution entworfen wurde. Dabei werden einzelne Lösungen durch ihre Eigenschaften bezüglich den Selektionsbedingungen beschrieben. Nur solche Lösungen, die sich als möglichst geeignet behauptet dürfen ihre Eigenschaften an ihre Nachfahren vererben. Im Laufe mehrerer Durchläufe entwickelt sich so die „Population“ immer näher in Richtung der optimalen Lösung oder der Lösungsmenge.

Die wichtigsten Ansätze der evolutionären Algorithmen lassen sich auf drei biologische Prinzipien zurückführen: Mutation, Rekombination und Selektion. Im Allgemeinen lässt sich die Vorgehensweise der EA in folgende Schritte aufteilen:

1. *Initialisierung:* Zuerst wird die erste Generation aus einer ausreichend großen Menge unterschiedlicher „Individuen“ (möglichen Lösungen) zufällig gebildet.

2. *Evaluation:* Für jeden Lösungskandidaten wird anhand einer sog. Fitness-Funktion seine Güte bezüglich der nicht-dominierenden Lösungen bestimmt.

3. *Selektion:* Die Lösungskandidaten aus der aktuellen Generation werden zufällig für die Vererbung ausgewählt. Dabei werden die Lösungen mit einer höheren Güte bezüglich

der Fitness-Funktion mit einer größeren Wahrscheinlichkeit in die Auswahl übernommen.

4. *Rekombination* (Crossover): Die Eigenschaften, auch Genome genannt, der ausgewählten Lösungskandidaten werden gemischt und an neue Lösungen weitergegeben (Vererbung).

5. *Mutation:* Die entstandenen Nachfahren werden zufällig verändert. Dabei hängt von der Stärke der Veränderung der einzelnen Genome ab, wie schnell sich die Lösungen den gesuchten Optimum annähern können.

6. Aus den neuen (und je nach Verfahren auch aus den alten) Lösungskandidaten werden die Mitglieder der neuen Generation ausgewählt. Hierbei werden bevorzugt „gute“ Lösungen übernommen. Diese Generation wird dann zur aktuellen.

7. Bei Erfüllung eines Abbruchkriteriums werden die besten Lösungskandidaten ausgegeben und der Algorithmus beendet. Sonst wird die Berechnung weiter mit Schritt 2 fortgeführt.

Der große Vorteil von solchen Algorithmen bei der Berechnung der Pareto-optimalen Lösungsmenge besteht in der Möglichkeit, in einem Durchlauf gleich mehrere Lösungen zu finden. Klassische Methoden dagegen finden nur eine Lösung pro Durchlauf. Die gefundenen Lösungen werden in jeder Iteration durch die Algorithmen so lange verändert, bis die gewünschten Voraussetzungen erreicht werden. Zum einen sollen die gefundenen Pareto-dominanten Ergebnisse möglichst nah an der realen Pareto-Front liegen, zum anderen sollte die Streuung entlang der Pareto-Front maximiert werden.

Ein sehr bekannter evolutionärer Algorithmus, der zur Bestimmung von Pareto-optimalen Fronten benutzt wird ist der *NSGA II*-Algorithmus (Abk. für *Non-Dominated Sorting Genetic Algorithm*) [Deb01]. An diese Stelle wird eine kurze Beschreibung gegeben, für eine genaue Beschreibung sei auf die Originalliteratur verwiesen.

NSGA II - Algorithmus

Die Abbildung 4.5 illustriert die Arbeitsweise des NSGA II - Algorithmus. Die Nachfahren-Population (Größe N) wird durch die Anwendung der üblichen Operationen Selektion, Rekombination und Mutation auf die Eltern-Population (Größe N) erzeugt. Die erzeugte Nachfahren-Population wird mit der Eltern-Kombination kombiniert, wodurch eine Population der Größe $2N$ entsteht. Anschließend wird die gesamte kombinierte Population bezüglich der Nicht-Dominanz sortiert und in mehrere nicht-dominierende Fronten $F_1, F_2, \ldots$ aufgeteilt. Eine neue Population der Größe N ist erzeugt, indem die Mitglieder der kombinierten Population aus unterschiedlichen Fronten übernommen werden. Dabei werden die Mitglieder aus den „besseren“ Fronten bevorzugt übernommen und andere nicht-dominierende Fronten verworfen. Da unter Umständen auch nicht alle Mitglieder der

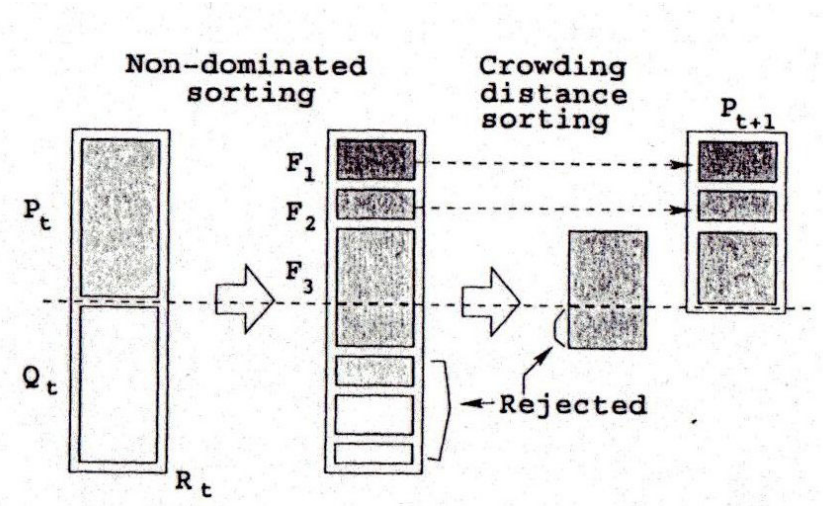


Abbildung 4.5.: NSGA II [Bur05]

letzten übernommenen Front in die neue Population passen, wird nur eine entsprechende Anzahl der Lösungskandidaten aus der letzen Front übernommen, basierend auf der *Crowding Distance* der einzelnen Lösungen.

Die Crowding Distance ist ein Maß für die Streuung der Lösungen entlang der Pareto-optimalen Front und wird als die Summe der Abstände zu den direkt benachbarten Lösungen für jedes Kriterium berechnet. Die Lösungskandidaten mit den größten Crowding Distance Werten werden dabei bevorzugt übernommen.

Der Vorteil des Algorithmus liegt in der guten Konvergenz gegen die Pareto-optimale Front. Durch die bevorzugte Behandlung der nicht-dominierten Lösungen wird sichergestellt, dass in jeder Population gute Lösungen überleben und sich somit fortpflanzen können.

4.2. Darstellung der Kriterien im Compiler

Damit ein Compiler die allgemein formulierten Forderungen nach optimierter Laufzeiteffizienz, Programmgröße oder Batterielaufzeit umsetzen kann, müssen diese Kriterien in messbaren Einheiten dargestellt werden. Anhand dieser messbaren Größen können anschließend die Optimierungen durchgeführt werden, die die einzelnen Größen minimieren. So lässt sich die Forderung nach Echtzeitschranken für harte und weiche Echtzeitsysteme gut durch den Begriff der maximalen Ausführungszeit (WCET) beschreiben. Die allgemeine Laufzeiteffizienz des Programms lässt sich dagegen durch die durchschnittliche Ausführungszeit (ACET) darstellen. Der Energieverbrauch des Systems beeinflusst direkt die Laufzeit der Batterien, da ein niedriger Verbrauch die Laufzeit steigert.

Im Folgenden werden die einzelnen Kriterien beschrieben, deren Betrachtung durch die Erweiterung des Compiler-Frameworks vom WCC ermöglicht wurde. Die realisierten Analysen sollen dabei allen Architekturen zur Verfügung stehen, die durch das retargierbare

Compiler-Backend vom WCC vom Compiler unterstützt werden.

4.2.1. WCET

Der Begriff Worst-Case Execution Time (WCET) bezeichnet die maximale Ausführungszeit eines Programms über alle möglichen Eingabedaten hinweg. Die genaue Ermittlung der WCET ist in im allgemeinen Fall unentscheidbar, da das Problem der Berechnung der WCET sich leicht auf das Halteproblem reduzieren lässt. Lässt sich die exakte WCET eines Programms bestimmen, so ist diese endlich ($WCET < \infty$). Somit muss das Programm spätestens nach Ablauf der WCET terminieren. Alan Turing hat bereits 1936 bewiesen, dass das Halteproblem unentscheidbar ist, d.h. für ein Programm P ist es nicht entscheidbar, ob dieses terminieren wird [Weg05]. Die exakte Bestimmung der WCET würde diese These widerlegen, somit ist eine genaue Berechnung der WCET im allgemeinen Fall nicht möglich. Es kann also lediglich approximative Lösungen geben.

In der Literatur wird deswegen zwischen der $WCET_{real}$ und der $WCET_{est}$ unterschieden. Dabei steht $WCET_{real}$ für die reale, maximale Laufzeit eines Programms, die im Allgemeinen unbekannt ist. Die $WCET_{est}$ hingegen bezeichnet die approximierte WCET, die durch Laufzeitanalysen gewonnen wurde. Da die maximale Laufzeit des Programms oft stark von den Eingabedaten abhängt, kann die $WCET_{est}$ sich erheblich von der $WCET_{real}$ unterscheiden. Zwei wichtige Voraussetzungen, die jedes Verfahren zur Bestimmung der $WCET_{est}$ beachten soll sind Sicherheit und Präzision. Eine Abschätzung der realen WCET heißt genau dann *sicher*, wenn sie mindestens so groß wie die reale WCET ist, wenn also gilt:

$$WCET_{est} \geq WCET_{real} \tag{4.1}$$

Diese Bedingung folgt aus dem Wunsch, die sicherheitsrelevanten Kriterien einzuhalten. Wenn die Gültigkeit dieser Ungleichung nicht garantiert ist, kann die approximierte WCET kleiner als die reale abgeschätzt werden. Das betrachtete Programm kann dann unter Umständen die vorgegebenen Echtzeitschranken nicht einhalten. Gerade bei harten Echtzeitsystemen kann dies zu katastrophalen Folgen führen.

Die zweite Bedingung verlangt, dass die Approximation der realen WCET *präzise* sein soll:

$$WCET_{est} - WCET_{real} \rightarrow min \tag{4.2}$$

Die Abweichung zwischen der approximierten $WCET_{est}$ und der realen $WCET_{real}$ sollte somit möglichst gering sein, da ansonsten die durch Abschätzung gewonnene Ergebnisse aus wirtschaftlichen Gründen nur bedingt nutzbar sind. Die Hardware, auf der das entsprechende eingebettete System aufgebaut ist, wird möglichst optimal unter den Aspekten des Stromverbrauchs, Leistung etc. ausgewählt. Ist die Präzision der Approximation zu niedrig, wird sich die Wahl der Hardware stark von der optimalen unterscheiden. Wenn z.B. die abgeschätze $WCET_{est}$ doppelt so groß wie die $WCET_{real}$ ist, so wird für die Einhaltung der gleichen Echtzeitschranke ein Prozessor benötigt, der doppelt so schnell arbeitet. Solche Prozessoren sind in der Regel teurer und verbrauchen meist mehr Strom, was sich negativ auf den Eigenschaften des Gesamtsystems auswirkt.

Aus diesen Gründen sollten die Verfahren zur Bestimmung der $WCET_{est}$ stets eine sichere und möglichst präzise Abschätzung der realen WCET liefern. Bei solchen Verfahren wird zwischen drei Ansätzen unterschieden, der statischen und dynamischen WCET-Analyse und dem hybriden Ansatz. Diese Ansätze werden im Folgenden vorgestellt.

Statische WCET-Analyse

Die statische WCET-Analyse verzichtet auf Auswertung der Daten aus Testläufen des Programms. Das zu analysierende Programm muss also zu keinem Zeitpunkt der Analyse auf einer Zielarchitektur ausgeführt werden. Stattdessen wird versucht, über die statistische Analyse des Maschinencodes des Programms eine sichere obere Schranke für die WCET anzugeben, die bei einem korrekten Modell des Verfahrens die Gleichung 4.1 erfüllt.

Dazu wird der Maschinencode in einen Kontrollflussgraph (engl. *Control Flow Graph*, Abk. *CFG*) transformiert. Der Kontrollflussgraph $CFG = (V, E)$ ist ein gerichteter Graph, der die Ausführungsreihenfolge eines Programms P repräsentiert. Dabei bezeichnet V die Menge aller Basisblöcke und E die Menge aller Kontrollusskanten. Ein Basisblock ist nach Definition auf Seite 28 eine maximale Menge von direkt aufeinanderfolgenden Anweisungen, die im Programmablauf nur an der ersten Anweisung betreten und nur an der letzten wieder verlassen werden können. Eine gerichtete Kante $e_{b1,b2}$ wird genau dann in den Graphen eingefügt, wenn es einen möglichen Kontrollfluss des Programms gibt, der direkt nach der Ausführung von Basisblock $b1$ in den Basisblock $b2$ wechseln kann.

Die Abbildung 4.6 zeigt einen kurzen Code-Abschnitt und den dazugehörenden Kontrollflussgraphen. Wie deutlich sichtbar wird, beschreiben die gerichteten Kontrollflusskanten die unterschiedlichen Ausführungspfade im Programm. Der Kontrollflussgraph stellt die grundlegende Datenstruktur der statischen WCET-Analyse dar und wird zur Beschreibung der möglichen Ausführungspfade des Programms verwendet.

Nach der Aufstellung des Kontrollflussgraphen für das gesamte Programm wird die $WCET_{est}$ für jeden einzelnen Basisblock bestimmt. Die Kosten eines Basisblocks ergeben sich dabei aus der Anzahl und Kosten der im Basisblock enthaltenen Befehle. Des Weiteren werden für komplexe zu analysierende Programmen weitere Benutzerangaben benötigt, da das zugrundeliegende Problem der WCET-Berechnung unentscheidbar ist. Es handelt sich dabei um Zusatzinformationen wie Schleifen- oder Rekursionsgrenzen, die für das jeweilge Verfahren zwingend notwendig sind, um die $WCET_{est}$ auszurechnen. Die Bestimmung von Schleifengrenzen kann dabei durch eine Schleifenanalyse wie in [LCFM09] geschehen, oder durch den Benutzer, der in diesem Fall dann die Verantwortung für die Korrektheit der Angaben übernimmt. Die Güte und Menge der bereitgestellten Zusatzinformationen bestimmt dabei die Qualität der Ergebnisse nach der Gleichung 4.2 mit.

Als zusätzliche Anforderung benötigt die statische WCET-Analyse ein abstraktes Modell der zugrundeliegenden Zielarchitektur. Die Verwendung von Caches erfordert die Aufstellung eines sehr genauen Cache-Modells, um die Kosten für die Speicherzugriffe möglichst präzise abschätzen zu können. Zusätzlich besitzen moderne Prozessoren eine oder mehrere Pipelines zur parallelen Ausführung von Instruktionen. Für eine gute Approximation der $WCET_{real}$ müssen die Pipelines ebenfalls genau modelliert und mit in die Berechnung

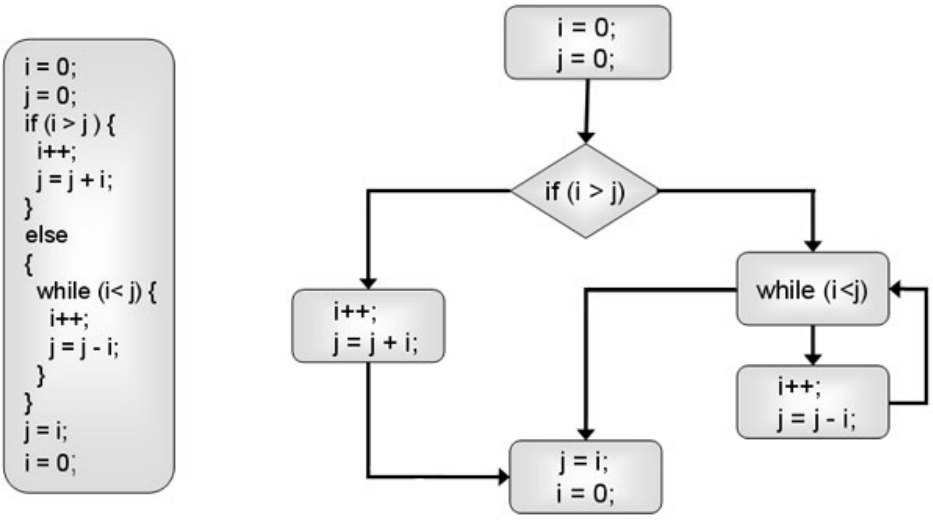


Abbildung 4.6.: Beispiel für einen Kontrollflussgraphen

einbezogen werden.

Zuletzt wird mithilfe des mit $WCET_{est}$-Informationen annotierten Kontrollflussgraphen der längste Ausführungspfad (engl. *Worst-Case Execution Path*, Abk. *WCEP*) berechnet. Die WCET eines Programms entspricht dabei der Länge des WCEP. Dafür werden unterschiedliche Verfahren eingesetzt, eines ist z.B. das *Implicit Path Enumeration Technique* (Abk. *IPET*). Hierbei wird die $WCET_{est}$ eines Programms als ein Optimierungsproblem betrachtet:

$$WCET_{est} = C(P) = \sum_{i}^{N} c_i \cdot x_i$$

Dabei bedeutet c_i die Kosten, die bei der Ausführung des Basisblocks b_i entstehen und x_i die Ausführungshäufigkeit des Basisblocks. N bezeichnet die Menge aller Basisblöcke. Zusätzlich werden Nebenbedingungen aufgestellt, die den Kontrollfluss wiedergeben, so muss die Ausführungshäufigkeit eines Basisblocks gleich der Summe der Ausführungen seiner direkten Vorgänger sein. Insgesamt entsteht so ein Gleichungssystem, dass z.B. durch ganzzahlige lineare Programmierung gelöst werden kann. Neben IPET gibt es noch andere Verfahren zur Berechnung von WCEP wie Syntaxbaum- oder Pfadbasierte Methoden, diese werden hier nicht weiter vorgestellt.

Ein Beispiel für eine ausgeklügelte statische WCET-Analyse bietet das in Abschnitt 2.2.3 beschriebene Analysetool aiT von AbsInt.

Dynamische WCET-Analyse

Im Gegensatz zu der statischen WCET-Analyse wird bei der dynamischen Analyse ein experimenteller Ansatz verfolgt, bei dem die $WCET_{est}$ durch Simulation abgeschätzt wird.

Dazu wird die ausführbare Binärdatei mehrmals mit unterschiedlichen Eingaben auf der Zielplattform (oder einem Simulator) ausgeführt. Durch die vielen Durchläufe soll die starke Abhängigkeit der WCET von den Eingabedaten relativiert werden. Die erhaltenen Ergebnisse der einzelnen Probeläufe werden anschließend für die Approximation der $WCET_{real}$ analysiert.

Allerdings sind die Eingabedaten, die zur maximalen Ausführungszeit führen meistens nicht bekannt. Um trotzdem eine repräsentative Abschätzung zu erhalten müssen möglichst viele unterschiedliche Kombinationen der Eingabevektoren getestet werden. Da bei einem Simulationsdurchlauf nur ein einzelner Kontrollflusspfad getestet werden kann ist das Ausprobieren aller möglichen Eingaben bei komplexeren Programmen nicht in vertretbarer Zeit möglich.

Der Nachteil der dynamischen WCET-Analyse ist, dass für die einzelnen Kontrollflusspfade gilt:

$$WCET_{est} \leq WCET_{real}$$

Dies steht im direkten Widerspruch zur Gleichung 4.1, die Bedingung der Sicherheit des Berechnungsverfahren ist somit verletzt. Durch viele Testläufe erhöht sich die Wahrscheinlichkeit, dass die $WCET_{est}$ nah an $WCET_{real}$ geschätzt wird, im allgemeinen Fall aber kann nicht bestimmt werden, ob die Sicherheitsregel verletzt wurde.

Der große Vorteil der dynamischen WCET-Analyse ist jedoch die unkomplizierte und schnelle Umsetzung des Verfahrens in die Praxis, weshalb es trotz aller Nachteile immer noch in der Industrie eingesetzt wird. Im Gegensatz zu der statischen WCET-Analyse wird hier kein präzises mathematisches Modell der Zielarchitektur mit den Zusatzinformationen gefordert, lediglich das Programm selbst und die Zielplattform.

Das größte Problem der dynamischen WCET-Analyse bleibt die Erzeugung von repräsentativen Eingabevektoren. Für die Auswahl von geeigneten Eingaben werden verschiedene Strategien verwendet, wie z.B. randomisierte Verfahren oder evolutionäre Algorithmen, um möglichst viele Testfälle abzudecken.

Hybride WCET-Analyse

Zusätzlich zu den beiden vorgestellten Ansätzen gibt es noch hybride Methoden. Diese versuchen die Vorteile beider Analyseansätzen zu vereinen, um die Präzision eines dynamischen Verfahrens durch statische Komponenten zu erhöhen.

Dafür werden in dem Kontrollflussgraphen die Laufzeiten einzelner Teilpfade durch Simulation ermittelt und in die Berechnung des statischen Verfahren integriert, wie z.B. in die Methode der ganzzahligen linearen Programmierung bei dem IPET-Verfahren. Allerdings entstehen dabei die typischen Probleme der dynamischen WCET-Analyse, da für die simulierte Teilpfade keine sichere WCET mehr garantiert werden kann.

Die Idee des hybriden Ansatzes besteht in der Hoffnung, durch die Simulation von großer Anzahl der Teilpfade das Verhalten von Caches oder Pipelines möglichst einfach in das Verfahren einbringen zu können. In einem rein mathematischen Modell, wie es in statischen Verfahren gefordert wird, ist das mit hohen Aufwand verbunden.

Ein Beispiel für hybrides Verfahren bietet SymTA/S [sym10]. Dieses Verfahren ermittelt durch Simulation die Laufzeiten einzelner zusammenhängender Blöcke ohne Verzweigungen, um sie für komplexere Verzweigungen in das verwendete mathematische Modell zu integrieren.

4.2.2. ACET

Der Begriff Average-Case Execution Time (ACET) bezeichnet die durchschnittliche Ausführungszeit eines Programms. Unter durchschnittlicher Ausführungszeit wird dabei die erwartete Laufzeit, die das Programm zum Abarbeiten einer zufällig und gleichverteilt gewählten Permutation aus der Menge aller möglichen Eingaben benötigt. Da eine solche durchschnittliche Eingabe sich nur schwer definieren lässt, wird in der Literatur unter ACET oft die *asymptotische Laufzeit* verstanden. Dies ist die Zeit, die das Programm im Durchschnitt über alle möglichen Eingabedaten hinweg zur Ausführung benötigt. Anders als bei WCET wird hier also nicht nach einem längsten Fall gesucht, sondern nach dem typischen Verhalten des Programms. ACET und WCET des gleichen Programms können sich daher stark unterscheiden.

Die numerische Untersuchung des asymptotischen Verhaltens von Algorithmen und Datenstrukturen ist schon seit Langem ein viel bearbeitetes Forschungsfeld. Ein breiter Überblick über die ACET-Analyse von vielen Standardalgorithmen ist z.B. in [Knu73] gegeben. Ein exaktes mathematisches Modell für komplexe Programme unter Beachtung der Zielarchitektur zu erstellen ist nur unter sehr großem Aufwand möglich. Daher wird bei der ACET-Analyse von Programmlaufzeiten oft der experimentelle Weg genommen, bei dem das Programm in mehreren Testläufen mit unterschiedlichen Eingaben ausgeführt wird. Die dabei gesammelten Laufzeitinformationen werden anschließend für die Bestimmung der ACET verwendet.

An dieser Stelle sollen noch etwas genauer die Methoden für die experimentelle Ermittlung der ACET unterschieden werden. Um die Ergebnisse der Testläufe zu erhalten, kann das zu untersuchende Programm entweder auf der real existierenden Hardware oder auf einem Simulator ausgeführt werden. Die Ausführung auf der Hardware bringt dabei mehrere Nachteile mit sich. Erstens ist die Messung der Ausführungszeiten oft kompliziert, da eingebettete Systeme meistens keine Peripherie aufweisen, an der die verstrichene Zeit leicht ausgegeben werden könnte. Somit müssen für den Testvorgang spezielle Vorrichtungen, wie z.B. Stoppuhren, Messgeräte u.Ä. eingesetzt werden. Zweitens kann das Programm so nur auf einer bereits existierenden Zielplattform getestet werden, was keine Rückschlüsse über das Verhalten auf anderen Plattformen erlaubt, die sich z.B. noch in Entwicklung befinden und somit nur als Beschreibung vorliegen.

Aus diesem Grund werden die Messungen oft an Simulatoren ausgeführt. Hierbei wird das zugrundeliegende Zielarchitektur mit allen Eigenschaften wie Befehlssatz, vorhandene on- und offchip-Speicher, Pipelines, Bussen usw. modelliert. Als Ausgabe wird meistens eine Art von Trace-Datei generiert, die Angaben wie Mnemonics und Speicheradressen von ausgeführten Instruktionen, Speicherzugriffe, Cache-Hits oder -Misses, benötigte Prozessorzyklen etc. beinhaltet. Damit lassen sich die Testläufe schnell und unkompliziert ausführen.

Im Wesentlichen werden zwei Arten von Simulatoren unterschieden, die *eventgesteuerten* und die *zyklengesteuerten*. Erstere ändern den System- bzw. den Programmzustand nur dann, wenn ein neues Ereignis auftritt, z.B. wenn eine neue Eingabe vorliegt oder wenn die CPU mit der Abarbeitung einer Instruktion fertig ist. Die interne Zeit des Simulators wird dabei um die entsprechend verstrichene Zeit erhöht. Zyklengesteuerte Simulatoren dagegen arbeiten in der Echtzeit und simulieren bei der Abarbeitung eines Programms jeden einzelnen Taktzyklus des Prozessors.

Für die ACET-Ermittlung der mit dem WCC kompilierten Programmen wird das Entwicklungstool CoMET der Firma VaST Systems Technology [vas10] eingesetzt, das im folgenden Abschnitt vorgestellt wird. Anschließend wird im Abschnitt 4.2.2 die Berechnung der ACET im WCC beschrieben.

CoMET

CoMET ist ein Entwicklungswerkzeug, das dem Entwickler das Erzeugen von virtuellen Systemprototypen (Abk. *VSP*) für komplexe eingebettete Systeme ermöglicht. Damit kann eine simultane Entwicklung, Optimierung und Simulation der Hardware und Software von solchen Systemen betrieben werden.

Zu dem Funktionsumfang von CoMET gehört eine Bibliothek mit einer Vielzahl von generischen Komponenten, die eine Hardware-Architektur ausmachen. Dazu gehören verschiedene Speicher, Timer, Interrupt-Controller u.Ä. Unterschiedliche Parameter wie Speicherzuordnung, Größe und Art der Caches oder Timing von Bussen können direkt eingestellt werden, um so ein Modell der gewünschten Architektur aufzubauen. Mitgeliefert wird eine Reihe bereits vorgefertigter virtueller Modelle für unterschiedliche Prozessoren, unter Anderem für den ARM7TDMI und für den Infineon TriCore TC1797. Somit ist eine Simulation von Programmen für diese Architekturen schnell möglich.

Die Simulation läuft eventgesteuert ab. Als Eingabe verarbeitet der Simulator eine Binärdatei für die entsprechende Architektur und produziert als Ausgabe eine Trace-Datei, die verschiedene benutzerdefinierte Metriken des simulierten Programm enthält. Pro Ereignis wird in der Trace-Datei eine Zeile mit der Nummer der entsprechenden Metrik (der sogen. *Record ID*), die das Ereignis ausgelöst hat, generiert. Die aufgezeichneten Informationen werden in dieser Zeile notiert. So hat z.B. der Trace-Eintrag für eine ausgeführte Instruktion das folgende Format:

Simulatorpfad	**Record ID**	**Nr.**	**Zyklus**	**Adresse**	**Opcode**	**Mnemonik**
`/Top/ArmCore/`	`Trace: T:1`	`4`	`110`	`0x10400360`	`0x1C3A`	`BX R1`

Tabelle 4.1.: Trace-Eintrag für eine Instruktion

Für die Berechnung von ACET (und später für Energieverbrauch) müssen folgende Metriken aufgezeichnet werden:

- `T:1` - Wird ausgelöst, wenn eine Instruktion ausgeführt wird. Weitere Informationen werden im nachfolgenden Beispiel aufgelistet. Von links nach rechts nach der Num-

mer der Metrik ist das die fortlaufende Instruktionsnummer, die seit dem Start des Programms verstrichene Zeit in Zyklen, die Adresse der Instruktion, den Opcode der Instruktion und die Assembler-Mnemonik der Instruktion. Die Metrik hat folgendes Format:

```
/Top/ArmCore/ Trace: T:1  4  110  0x10400360  0x1C3A  BX R1
```

- `T:8` - Wird ausgelöst, wenn eine Load- oder Store-Instruktion auf den Speicher zugreift. Weitere Informationen beinhalten die Speicheradresse, auf die zugegriffen wurde (`MVA` steht hierbei für *modified virtual address*). Diese Metrik betrachtet nicht die Instruktion-Fetches oder Cache-Misses und hat das folgende Format:

  ```
  /Top/ArmCore/ Trace: T:8  MVA 0x00000354
  ```

- `T:47` - Wird ausgelöst, wenn ein Cache-Miss beim Lesen von Daten- oder Instruktion-Cache vorliegt. Weitere Informationen beinhalten den Cache-Namen, Cache-Tag, Cache-Set und Cache-Way. Die Metrik hat das folgende Format:

  ```
  /Top/ArmCore/ Trace: T:47  Inst Cache Read Miss, Allocate
  Line, Tag 0x00000120, Set 4, Way 0
  ```

- `T:48` - Wird ausgelöst, wenn ein Cache-Miss beim Schreiben von Daten-Cache vorliegt. Weitere Informationen beinhalten den Cache-Namen, Cache-Tag, Cache-Set und Cache-Way. Die Metrik hat das folgende Format:

  ```
  /Top/ArmCore/ Trace: T:48  Data Cache Write Miss, Allocate
  Line, Tag 0x00000354, Set 2, Way 0
  ```

ACET-Berechnung im WCC

Für die Berechnung der ACET wurde im WCC eine eigene Bibliothek mit dem Namen *LIB-PROFILE* erstellt. Diese beinhaltet die Dateien *llirprofile.cc* und *llirprofile.h*, welche die Klasse LLIRProfile implementieren. Diese Klasse übernimmt die Berechnung der ACET eines mit dem WCC kompilierten Programms. Dazu muss der WCC mit der Kommandozeilenoption `-Oacet` aufgerufen werden, wodurch der `PROFILING`-Zustand im FSM-Automaten eingestellt wird (vergl. Abschnitt 3.2.2).

Die berechneten ACET-Werte werden in Objekten der Klasse LLIR_ACET_OBJ abgespeichert, die von der Klasse LLIR_Objective abgeleitet ist. Solche Objekte werden an jeden Basisblock und jede Funktion in der LLIR angehängt, sowie an die LLIR selbst. Über die folgenden Funktionen können die ACET-Werte abgefragt werden:

- *getACET()* - Diese Funktion gibt die globale ACET des annotierten LLIR-Konstruktes (LLIR_BB, LLIR_Function oder LLIR) zurück.

- *getACCallFrequency(LLIR_Function * f)* - Diese Funktion gibt für eine annotierte LLIR_Function die Aufrufhäufigkeit einer anderen Funktion `f`, die von der annotierten Funktion aufgerufen wird. Ruft die annotierte Funktion die andere Funktion `f`

nicht auf, dann wird 0 zurückgegeben. Die Abbilldung 4.7 links veranschaulicht die Beziehung. Die Funktion `main()` ruft die Funktion `foo()` 5 mal und die Funktion `calc()` 3 mal auf. Somit gibt der Aufruf von *getACCallFrequency(foo)* an `main()` „5“ und *getACCallFrequency(calc)* „3“ zurück.

- *getACECSum()* - Diese Funktion gibt für einen annotierten LLIR_BB die über die gesamte Programmausführung aufsummierte Ausführungshäufigkeit (engl. *Average Case Execution Count*) des Blocks zurück. Anhand der Abbilldung 4.7 rechts kann die ACEC-Berechnung nachvollzogen werden. Der Basisblock `L15` ruft sich selbst 5 mal auf, und wird noch weitere 2 mal von `L10` aufgerufen. Somit wird in der LLIR_ACET_OBJ von `L15` die ACECSum „7“ annotiert und von `L11` entsprechend „9“.

- *getACEC(LLIR_BB* bb)* - Diese Funktion gibt für einen annotierten LLIR_BB die ACEC seines Nachfolgers `bb` zurück, wenn es einen solchen Nachfolger gibt. Sonst wird 0 zurückgegeben. In diesem Fall wird nicht die globale ACEC betrachtet, sondern nur die Gesamtanzahl der Aufrufe des Nachfolger-BBs durch den annotierten Basisblock. Die Abbilldung 4.7 rechts illustriert die Berechnung. Der Basisblock `L11` wird 1 mal vom `L10` aufgerufen und weitere 8 mal von anderen Knoten des Kontrollflussgraphen aus. Somit ist in der LLIR_ACET_OBJ von `L10` für den Nachfolger `L11` die ACEC „1“ annotiert. Entsprechend wird für den Nachfolger `L15` die ACEC „2“ annotiert.

- *getSuccessorACECs()* - Diese Funktion gibt für einen annotierten LLIR_BB eine `std::map` mit allen seinen Nachfolgern und deren ACEC's (berechnet wie bei *getACEC()*) zurück.

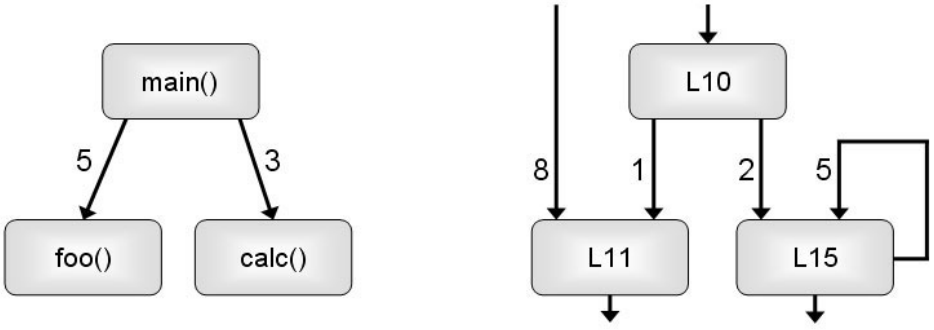


Abbildung 4.7.: Die Funktionen getACCallFrequency() und getACEC()

Die Klasse LLIRPRofile verarbeitet die vom CoMET erzeugte Trace-Datei, um die ACET-Informationen zu extrahieren. Dazu wird ein Textparser auf einer kontextfreien Grammatik eingesetzt, der in einer while-Schleife die einzelnen Zeilen der Trace-Datei

einliest und verarbeitet. Dabei werden die ausgeführten Instruktionen über ihre Speicheradressen auf die Zugehörigkeit zu den entsprechenden Basisblöcken getestet. Da die ACET-Informationen auf der Basisblockebene annotiert werden, werden die Ausführungszeiten für die einzelnen Instruktionen aufaddiert und an den jeweiligen Basisblock angehängt. Der folgende Pseudocode veranschaulicht die Arbeitsweise des Parsers:

```
1  var curr_bb_start = -1;
2  var curr_bb_end = -1;
3  Get new line;
4  if (line == file end) STOP;
5  if (Record ID != T:1) goto 3;
6  inst_memaddress = getInstrAddr(line);
7  c_count = getCycle(line);
8  if (curr_bb_start < inst_memaddress <= curr_bb_end)
9    goto 3;
10 if (inst_memadress != knownBB_memaddress)
11   if (inst_memaddress == main_end) STOP;
12   Warning: Unknown BB!;
13   last_c_count = c_count - last_c_count;
14   UpdateACET(lastBB, last_c_count);
15   goto 3;
16 currBB = getBB(inst_memadress);
17 curr_bb_start = inst_memadress;
18 curr_bb_end = curr_bb_start + bb_size;
19 last_c_count = c_count - last_c_count;
20 UpdateACET(lastBB, last_c_count);
21 UpdateACECSum(currBB);
22 UpdateACEC(lastBB -> currBB);
23 lastBB = currBB;
24 last_c_count = c_count;
25 goto 3;
```

Listing 4.1: Trace-Parser in Pseudocode

Für jede Zeile mit einer Trace-Metrik wird die Record ID ausgelesen, bis eine Instruktionszeile mit Record ID `T:1` gefunden wird (Zeile 5). Durch den Vergleich mit der Start- und Endadresse des aktuellen Basisblocks wird ermittelt, ob die Instruktion zu dem aktuellen Basisblock gehört (Zeile 8). Ist dies der Fall, kehrt der Algorithmus zurück zum Anfang und betrachtet eine neue Zeile. Für eine Instruktion außerhalb des aktuellen Basisblocks wird überprüft, ob die Speicheradresse der Instruktion mit einer Anfangsadresse eines bekannten Basisblocks übereinstimmt (Zeile 10). Wenn kein passender Basisblock gefunden wurde, wird eine Warnung über einen unbekannten Basisblock ausgegeben, die ACET des verlassenen Basisblocks aktualisiert und der Algorithmus mit der nächsten Zeile

neu gestartet (Zeilen 12-15). Falls ein neuer bekannter Basisblock gefunden wurde, wird die Start- und Endadresse auf die entsprechenden Werte des neuen Basisblocks gesetzt, die gesamten ACET-Informationen aus der LLIR_ACET_OBJ aktualisiert und dann weiter mit einer neuen Zeile vom Anfang des Algorithmus fortgefahren (Zeilen 16-25). Der Algorithmus stoppt, wenn es keine weiteren Zeilen in der Trace-Datei mehr gibt oder wenn die `main`-Methode verlassen wurde. In beiden Fällen ist die Betrachtung des simulierten Programms im CoMET beendet.

Für eine real eingesetzte Software kann die zu lesende Trace-Datei aus mehreren Millionen Metrik-Zeilen bestehen. Das Parsen der Trace-Datei beansprucht somit den weitaus größeren Teil der Ausführungszeit bei der ACET-Berechnung. Daher beeinflusst jede Beschleunigung des Verfahrens direkt die Laufzeit der gesamten Berechnung. Aus diesem Grund wurden, um eine schnellere Laufzeit beim Parsen der Instruktionen zu garantieren, zuerst in einem extra Schritt die Start- und Endadressen aller in der LLIR enthaltenen Basisblöcke über die Klasse `LLIR_MemoryLayout` ermittelt. Die Start- und Endadressen werden zusammen mit den dazugehörenden Basisblöcken in einer separaten `std::map` abgelegt. Damit sind für den Vergleich aus Zeile 10 keine Funktionsaufrufe an die LLIR mehr notwendig, es muss lediglich die lokale Datenstruktur abgefragt werden, was wesentlich schneller erfolgt.

Ein weiterer Optimierungsansatz entsteht aus der Beobachtung, dass in einem Programm in der Regel die Zeile 8 am häufigsten ausgeführt wird. Instruktionen, die nicht zu einem der Basisblöcke aus der LLIR gehören werden entweder vor dem Beginn der `main`-Methode vom Startup-Code oder im Programmfluss durch Systemaufrufe wie z.B. Interrupts erzeugt. Für eine effiziente Ausführung der 8. Zeile wird in Zeilen 17 und 18 beim Betrachten eines neuen Basisblocks seine Start- und Endadresse abgespeichert. Somit muss für alle nachfolgenden Instruktionen in diesem Basisblock nur ein einfacher Vergleich der Instruktionsadresse mit den beiden Adressen erfolgen, um zu bestimmen, ob eine Instruktion noch zu diesem Basisblock gehört.

Schließlich werden die ACET-Werte während der Verarbeitung der Trace-Datei in eine lokale Datenstruktur abgespeichert bzw. inkrementiert, und erst nach dem Beenden des Parsers an die jeweiligen Basisblöcke und Funktionen angehängt. Das hat den Vorteil, dass die Objekte der ACET-Objectives nur einmal pro Basisblock und Funktion erzeugt und mit den Werten aktualisiert werden müssen. Somit wirken sich Basisblöcke, die mehrmals in einer Schleife ausgeführt werden, nicht negativ aus die Laufzeit des Parsers aus.

4.2.3. Energie

Um den Energieverbrauch eines Programms zu definieren, müssen zunächst einige elektrotechnische Grundlagen eingeführt werden. Zuerst wird der Begriff der Leistung P eingeführt. Der Momentanwert der Leistung P ist als das Produkt aus der momentanen Spannung und dem momentanen Strom definiert:

$$P(t) = U(t) \cdot I(t)$$

$$\text{Einheit: } 1W \text{ (Watt)} = 1V \text{ (Volt)} \cdot 1A \text{ (Ampere)}$$

Die elektrische Energie ergibt sich aus dem Integral der Leistung über einen betrachteten Zeitraum $[t_0, t_1]$, z. B. der Ausführungszeit des Programms. Die Energie E, die durch einen elektrischen Verbraucher umgewandelt wird, wird berechnet durch:

$$E = \int_{t_0}^{t_1} U(t) \cdot I(t) dt = \int_{t_0}^{t_1} P(t) dt$$

$$\text{Einheit: } 1J \text{ (Joule)} = 1VAs = 1Ws$$

Werden Strom und Spannung über die Zeit gemittelt, dann kann die Gleichung wie folgt vereinfacht werden:

$$E = U \cdot I \cdot t = P \cdot t$$

Der Leistungsverbrauch in einer CMOS-Schaltung lässt sich in drei Ströme aufteilen: Leckstrom I_{lk}, Kurzschlussstrom I_{sc} und Schaltstrom I_{sw}.

- Der Leckstrom (engl. *leakage current*) fließt unabhängig von den Schaltvorgängen der Gatter. Im statischen Zustand der Schaltung, wenn an Eingängen keine Signaländerung stattfindet, ist dies der einzig fließende Strom. Die Hauptursache des Leckstroms ist der Tunnel-Effekt der Elektronen und Löchern in den Transistoren.

- Der Kurzschlussstrom (engl. *short circuit current*) fließt während eines Schaltvorgangs. Dabei entsteht für kurze Zeit eine gleichzeitige Leitfähigkeit der n-MOS und p-MOS-Transistoren, so dass der Strom von der Versorgungsspannung nach Masse fließt.

- Der Schaltstrom (engl. *switching current*) fließt während des Ladens und Entladens der Lastkapazitäten am Ausgang der Gatter. Die Kapazitäten werden umgeladen, wenn der Wert an der angelegten Leitung von 0 nach 1 und umgekehrt wechselt. Dieser Strom verursacht den weitaus größten Anteil an der Verlustenergie.

Die Verlustleistung einer CMOS-Schaltung ist proportional zu den vorhanden Kapazitäten C, der Schaltfrequenz α der Gatter und dem Quadrat der Spannung U [Mar07]:

$$P = \alpha \cdot C \cdot U^2 \cdot f$$

Energiemodelle

Damit der Compiler die energieminimierenden Optimierungen anwenden und evaluieren kann, muss der Energieverbrauch des kompilierten Programms bekannt sein. Da die physische Messung des Energieverbrauchs durch den Compiler nach jeder angewandter Optimierung nicht in die Frage kommt, erfolgt die Bestimmung durch ein Energiemodell.

Energiemodelle sind ein zentraler Bestandteil aller Energieoptimierungen. Energiemodelle werden benutzt, um zu den einzelnen Instruktionen oder Sequenzen der Instruktionen

den Energieverbrauch vorauszusagen. Im Allgemeinen besteht der Nachteil, dass solche Modelle oft nur eine begrenzte Genauigkeit besitzen. In [Mar07] wird ein Überblick der existierenden Modelle gegeben.

Eines der ersten Energiemodelle für Instruktionen wurde von Tiwari [TMW94] vorgeschlagen. Das Modell beschreibt die Basiskosten einzelner Instruktionen und die Inter-Instruktionskosten. Basiskosten beschreiben den Energieverbrauch, der bei jeder Ausführung der Instruktion entsteht, wenn die Instruktion unendlich oft in einer Schleife ausgeführt wird. Inter-Instruktionskosten entsprechen den zusätzlichen Energieverbrauch, der durch die Ausführung von unterschiedlichen Instruktionen entsteht. Dieser zusätzlicher Energieverbrauch ist z.B. auf das Ein- und Ausschalten von Funktionseinheiten wie z.B. ALU, FPU oder Shifter zurückzuführen. Dieses Energiemodell vernachlässigt den Energieverbrauch durch den Speicher oder weitere Komponenten des Systems.

Ein weiteres Energiemodell wurde von Lee [LEM01] vorgestellt. Hierbei wird die detaillierte Analyse der unterschiedlichen Effekte, die in Pipelines auftreten, beschrieben. Allerdings berücksichtigt dieses Modell keine Mehrzyklen-Instruktionen und keine Pipeline-Stalls.

Ein Energiemodell, das den Energieverbrauch von Caches berechnen kann ist CACTI [WJ96]. Es basiert auf einem analytischen Modell. Das Programm verarbeitet Eingabeparameter wie Cachegröße, Blockgröße, Assoziativität und Technologiegröße und liefert die Abschätzung für die notwendige Energie für die Cachezugriffe.

Das Energiemodell, das in dieser Diplomarbeit zur Bestimmung des Energieverbrauchs verwendet wird, wurde aus dem Analysetool enProfiler adaptiert. Es wurde am Lehrstuhl Informatik 12 der Technischen Universität Dortmund von Steinke et al. [SKWM01] entwickelt und verfolgt einen ähnlichen Ansatz wie das Energiemodell von Tiwari. Dieses Modell basiert auf genauen Messungen an realen Hardware. Insbesondere werden hier auch Energiekosten für die Zugriffe auf den Cache (über CACTI) und Speicher modelliert. Ebenfalls berücksichtigt werden die Hamming-Abstände zwischen den Daten und die Kosten für die Anzahl der Einsen auf den Bussen. Auch diese Daten wurden über genaue Messungen ermittelt. Wie bei dem Modell von Tiwari werden Basiskosten und Inter-Instruktions-Kosten angenommen.

Energieberechnung im WCC

Das Verfahren, das für die Berechnung des Energieverbrauchs des mit WCC kompilierten Programms angewendet wird, basiert auf dem Profiling-Ansatz aus 4.2.2. Die Klasse `LLIRProfile` wurde so erweitert, dass auch eine Betrachtung des Energieverbrauchs auf Basis der Trace-Informationen ermöglicht wird.

Die Grundidee des Parser-Algorithmus bleibt bestehen, es soll lediglich der Energieverbrauch durch die Speicherzugriffe zusätzlich zu den Instruktionen betrachtet werden. Dabei werden über die Adresse des Speichers, auf den zugegriffen wird, die Zugriffskosten ermittelt. Die Basiskosten werden über den Opcode-Feld der Metrik berechnet. Mithilfe eines Hash-Verfahren wird die Instruktionsklasse, zu der der entsprechende Opcode gehört, ermittelt und die Basiskosten für die Instruktion zurückgegeben.

Die Kosten für den Energieverbrauch der einzelnen Instruktionen und Speicherzugriffe wurden dem Energiemodell aus dem Analysetool EnProfiler entnommen. Dazu wird die Datenbank mit den gemessenen Energiekosten pro Instruktion und pro Speicherzugriff aus der Konfigurationsdatei des EnProfilers eingelesen. Da LLIRProfile einen eigenen Trace-Parser verwendet, ist die weitere Verwendung von EnProfiler nicht mehr notwendig.

Die berechneten Werte für den Energieverbrauch werden ähnlich wie bei ACET auf Seite 58 in der LLIR annotiert. Dazu wird eine weitere Klasse LLIR_ENERGY_OBJ von der Klasse LLIR_Objective abgeleitet. Über die Funktion *getEnergy()* kann hierbei der Energieverbrauch von einem LLIR_BB, LLIR_Function oder der gesamten LLIR abgefragt werden.

Das Verfahren lässt sich ohne Änderung für jede vom CoMET unterstützte Zielarchitektur einsetzen, greift aber auf die Werte des zugrundeliegenden Energiemodells zurück. Dieses Modell ist momentan nur für den ARM7TDMI-Prozessor vorhanden und müsste für alle anderen Zielarchitekturen neu erstellt werden.

4.2.4. Codegröße

Anders als die zuvor vorgestellten Kriterien lässt sich die Codegröße eines Programms relativ einfach bestimmen. Dazu muss die Größe der *.text*-Section des Programms berechnet werden. Dies ist der dem Programm zugeordnete Speicherbereich, wo die Instruktionen des Programms abgelegt werden. Eine Möglichkeit dazu stellt die LLIR mit der Funktion

```
int LLIR::GetCost()
```

dar. Diese Funktion gibt die Kosten für die gesamte LLIR zurück. Diese werden als die Summe der Kosten aller Instruktionen in der LLIR berechnet. Die Kosten für eine Instruktion spiegeln die für das Speichern des Befehls benötigte Anzahl von Bytes, also den Speicherplatz, den die Instruktion belegt. Über die Summe kann somit der Speicherplatz, der für das gesamte Programm benötigt wird, ermittelt werden.

5. Evaluierung

Die in den Kapiteln 3 und 4 vorgestellten Methoden und Mechanismen zur Realisierung eines retargierbaren Compiler-Backends und zur Unterstützung von multikriteriellen Optimierungen wurden im Rahmen der Diplomarbeit für den WCC implementiert. Um die entwickelten Mechanismen testen zu können, wird eine Reihe von Standardoptimierungstechniken bezüglich ihrer Auswirkungen auf die gemessene ACET sowohl für die TriCore- als auch für die ARM-Architektur untersucht. Dazu wurden aus den in ICD-C vorhandenen High-Level Analysen und Optimierungen bestimmte Optimierungsverfahren ausgewählt. Des Weiteren wurden die Auswirkungen auf die ACET und auf den Energieverbrauch beim Einsatz des ARM-Prozessors gemessen.

Um aussagekräftige Ergebnisse zu erhalten, wurden 70 Benchmarks aus der *WCET-BENCH*-Testsuite des WCC ausgewählt. Diese bestehen aus Benchmarks der Benchmarksammlungen DSPstone [vM94], MediaBench [LPMS97], MiBench [GRE+01] misc, MRTC [Mäl10], StreamIt [Str10] und UTDSP [UTD10]. Die misc-Benchmarksuite enthält dabei Benchmarks, die aus der ICD-C eigenen Testbench-Sammlung stammen. Diese Benchmarksammlungen enthalten Testprogramme aus den unterschiedlichsten Anwendungsbereichen wie Videokonvertierung, Sortieralgorithmen oder Signalverarbeitung. Somit können die implementierten Techniken in einem realitätsnahen Umfeld getestet werden. Eine genaue Auflistung aller verwendeten Benchmarks ist in Anhang A zu entnehmen.

Im Folgenden wird im Kapitel 5.1 eine kurze Übersicht über die Optimierungen gegeben, welche für die Messreihen eingesetzt werden. Kapitel 5.2 stellt die Ergebnisse der Testläufe für die ARM- und TriCore-Architektur vor. In Kapitel 5.3 werden die Messungen der ACET und des Energieverbrauchs auf der ARM-Architektur vorgestellt. In Kapitel 5.4 schließlich folgt die Auswertung der Ergebnisse. Die genaue Auflistung der Ergebnisse, die als Grundlage für die hier verwendeten Diagramme benutzt wurden, kann in Anhang A nachgelesen werden.

5.1. Verwendete Optimierungen

Die Ergebnisse für alle Benchmarks wurden zuerst separat für alle drei Optimierungsstufen `O1` bis `O3` des WCC bestimmt, mit Optimierungsstufe `O0` als Referenzwert. Mit jeder weiteren Optimierungsstufe werden dabei innerhalb des WCC eine größere Zahl an Optimierungen eingeschaltet. Pro Testfall, benutzte Architektur und eingeschaltete Optimierungsstufe wurde jeweils ein ACET-Wert gemessen, mit einer weiteren Messung für den Energieverbrauch auf der ARM-Plattform. Insgesamt wurden so 840 Messungen durchgeführt.

Neben den Messungen mit allen Optimierungsstufen wurden einige weitere Messungen durchgeführt. Das Ziel hierbei war die Untersuchung des Einflusses von einzelnen ausgewählten Standardoptimierungen. Auch hier wurde zwischen den zwei Architekturen verglichen und für die ARM-Architektur zusätzlich zwischen ACET und Energieverbrauch. Dazu wurden jeweils nur die einzelnen Optimierungen bei Optimierungsstufe `O1` eingeschaltet und gemessen. Anschließend erfolgte der Vergleich mit der Referenzstufe `O1`. Dadurch wurden weitere 1470 Ergebnisse gemessen. Die einzelnen ausgewerteten Optimierungen sind:

- *Loop Collapsing*: Diese Technik optimiert verschachtelte Schleifen, die auf multidimensionale Arrays zugreifen. Dabei werden die Schleifen zu einer einzigen Schleife zusammengefasst, die über die Länge des gesamten Array iteriert, anstatt nur jeweils über einzige Dimensionen. Das führt zu einer signifikanten Reduktion der notwendigen Vergleiche und Sprungbefehle.

- *Loop Deindexing*: Hierbei werden die Array-Zugriffe über Indizes innerhalb der Schleifen durch automatisch inkrementierte Load- und Store-Operationen ersetzt. Dadurch wird der Aufwand beim Zugriff auf Arrayelemente verringert.

- *Loop Head Transformation*: Bei dieser Optimierung werden sog. kopfgesteuerte (engl. *head controlled*) `while`- und `for`-Schleifen in fussgesteuerte (engl. *foot controlled*) `do-while`-Schleifen transformiert. Somit passiert die Prüfung der Schleifenbedingung erst nach der Ausführung, wodurch in jeder Schleife Sprungbefehle eingespart werden können.

- *Loop Unrolling*: Diese Optimierung dupliziert den Schleifenkörper, sodass die Instruktionen in der Schleife direkt mehrmals hintereinander ausgeführt werden. Dadurch wird der Aufwand für die Prüfung der Schleifenbedingung und die Anzahl der notwendingen Sprünge verringert. Vollständiges „Ausrollen“ der Schleifen führt zur völligen Reduktion des Overheads für die Prüfung der Schleifenbedingung und für die Sprünge, erhöht aber entsprechend die Codegröße.

- *Loop Unswitching*: Hierbei werden Verzweigungen innerhalb Schleifen, die unabhängig von Schleifenvariablen sind, außerhalb der Schleife platziert. Die Schleife wird kopiert und jeweils eine passende Version in die einzelnen Verzweigungen verschoben. Damit wird die Überprüfung der bedingten Anweisung nur einmal ausgeführt und nicht bei jeder Schleifeniteration.

- *Function Inlining*: Diese Optimierung fügt eine Kopie des Funktionsrumpfes der aufgerufenen Funktion direkt an die Stelle des Aufrufs ein. Die Aufrufparameter werden dabei in den kopierten Funktionsrumpf propagiert. Dadurch entfallen die notwendigen Sprungbefehle und Overhead für Function Call und Return. Durch denn Einsatz dieser Optimierung kann der Programmcode stark vergrößert werden. Deswegen werden nur solche Funktionen eingefügt, die eine bestimmte Größe nicht übersteigen.

- *Struct Scalarization*: Lokale struct-Objekte, die Bitfelder ausschließlich mit lokalen Zugriffen enthalten, werden in äquivalente Objekte des Typs Integer umgewandelt. Skalare Komponenten eines struct-Objekts nur mit direkten Zugriffen werden zu eigenständigen Komponenten transformiert. Das ermöglicht eine effizientere Zuordnung der Ressourcen bei der Codeerzeugung.

5.2. Vergleich der Einflüsse von Standardoptimierungen auf die ACET der Zielarchitekturen

Im diesen Abschnitt werden die einzelnen Messungen vorgestellt, die bei der Untersuchung der ACET-Ergebnisse für die oben beschriebenen Optimierungen entstanden sind.

5.2.1. Vergleich der Standardoptimierungsstufen

Zuerst werden die Auswirkungen der Optimierungsstufen `O0` bis `O3` auf die ACET der einzelnen Benchmark-Suiten für die beiden Architekturen untersucht. Abbildung 5.1 stellt die Ergebnisse dar. Auf der X-Achse sind die einzelne Optimierungsstufen und Benchmark-Suiten dargestellt. Die Y-Achse bildet die relative ACET ab. Innerhalb des Diagramms wird immer der Mittelwert der gemessenen relativen ACET über die einzelnen Benchmarks in jeder Benchmark-Suite dargestellt. Der Referenzwert von 100% entspricht dem Mittelwert der gleichen Benchmarks der jeweiligen Benchmark-Suite bei der Optimierungsstufe `O0`.

Insgesamt entspricht das gemessene Verhalten dem, was im allgemeinen Fall mit steigenden Optimierungsstufen erwartet wird: bis auf einige Ausreißer bei misc und UTDSP sinkt bei jeder höheren Optimierungsstufe der gemessene relative ACET-Wert. Gut sichtbar sind die Unterschiede der Zielarchitekturen, auf denen die Messungen durchgeführt waren. So zeigt der TC1797-Prozessor im Vergleich zu dem ARM7TDMI ein viel besseres Verhalten bei Benchmarks, die mit Fließkommazahlen arbeiten, vor allem DSPstone floating point. Im Gegensatz zu TC1797 besitzt der ARM7TDMI keine FPU für Fließkommaberechnungen und muss die entsprechenden Berechnungen über Software ausführen. Bei der Benchmark-Suite DSPstone fixed point dagegen, wo die gleichen Benchmarks mit Integer-Werten operieren ist der Unterschied zwischen den Architekturen nicht so deutlich ausgeprägt. Ein ähnlich großer Vorsprung bei der Berechnung kann auch für die Benchmark-Suite StreamIt beobachtet werden. Hierbei wird Signalverarbeitung simuliert, die durch die DSP-Funktionalität des TriCore-Prozessors effizienter durchgeführt wird. Die Vergrößerung der ACET für die Optimierungsstufe `O2` bei den Benchmark-Suiten UTDSP und misc (für TriCore) kann mit den Auswirkungen von Schleifenoptimierungen erklärt werden. Diese werden erstmals in der Optimierungsstufe `O2` eingeschaltet und erwirken für manche Benchmarks eine Verschlechterung der Laufzeit. Vor allem für die Optimierungen *Loop Deindexing* und *Loop Head Transformation* wurden bei der Messung der einzelnen Optimierungen in nachfolgenen Abschnitt für viele Benchmarks schlechtere ACET-Werte gemessen, als bei der Referenzstufe O1.

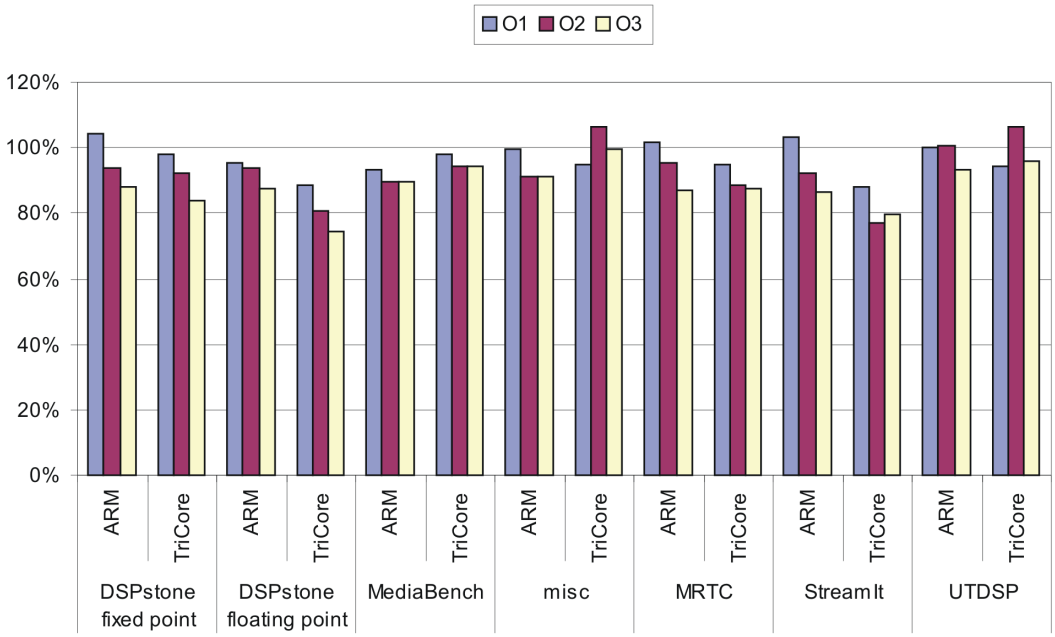


Abbildung 5.1.: Vergleich ACET-Reduktion bei unterschiedlichen Optimierungsstufen und Architekturen

Der beste Messwert für die relative ACET für ARM wurde mit 86,9% bei der Benchmark-Suite MRTC mit der Optimierungsstufe `O3` ermitellt, der schlechteste mit 86,9% bei DSPstone fixed point mit `O1`. Für TriCore wurde das beste Ergebnis mit 74,2% der relativen ACET bei DSPstone floating point mit der Optimierungsstufe `O3` gemessen und das schlechteste mit 106,5% bei UTDSP bei `O0`. Im Durschnitt wird bei `O1` eine relative ACET von 99,6% für ARM und 93,8% für TriCore erreicht, bei `O2` entsprechend 93,9% für ARM und 92,2% für TriCore und bei `O3` 89% für ARM und 87,9% für TriCore.

5.2.2. Vergleich der einzelnen Optimierungen

Anschließend folgt die Auswertung des Einflusses der in 5.1 ausgewählten Optimierungen auf die ACET der Zielarchitekturen ARM und TriCore. Pro Optimierung zeigt ein Diagramm die Messergebnisse auf den einzelnen Benchmarks. Da viele Standardoptimierungen und vor allem Schleifenoptimierungen erst nach weiteren Optimierungen wie *Common Subexpression Eliminaton* ihre volle Wirkung zeigen, ist die Benutzung der Optimierungsstufe `O0` in diesem Fall wenig sinnvoll. Deshalb wurde die Optimierungsstufe `O1` als Referenz gewählt, die der 100%-Linie in den folgenden Diagrammen entspricht. Hierbei werden nur die optimierten Benchmarks verglichen, bei denen eine deutlich messbare Abweichung von dem Referenzwert gemessen wurde (der gemessene Wert ist $> 101\%$ oder $< 99\%$). Der Durchschnittswert (letztes Balken mit dem Label *average*) wird über alle Benchmarks gebildet, einschließlich der nicht-optimierten. Die Y-Achse stellt immer die relative ACET

dar, mit den einzelnen Benchmarks auf der X-Achse.

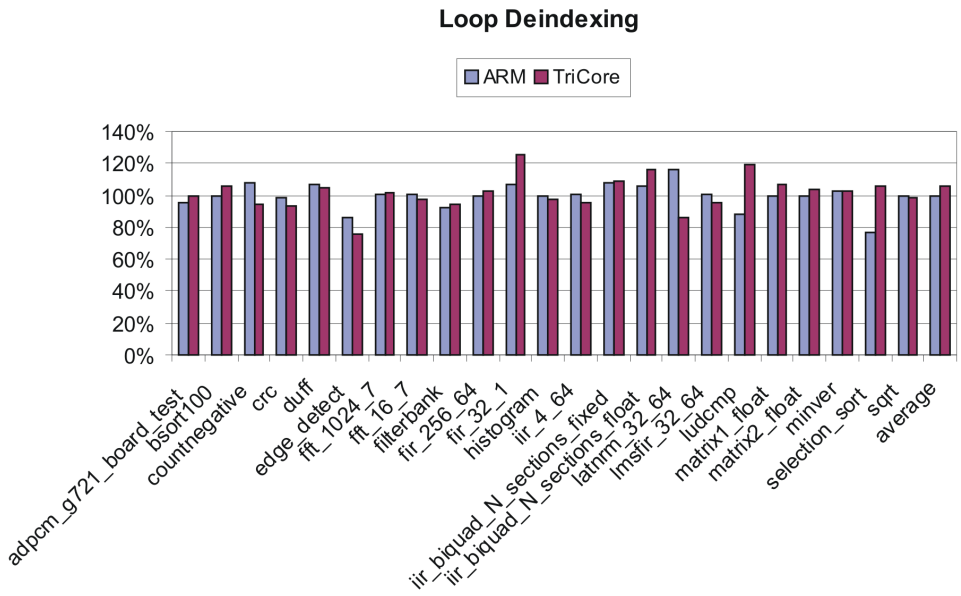


Abbildung 5.2.: Einfluss von Loop Deindexing auf die ACET unterschiedlicher Architekturen

Für die Optimierungen *Loop Collapsing*, *Loop Unswitching* und *Struct Scalarization* wurden über alle Benchmarks hinweg keine Fälle mit der ACET $> 101\%$ oder $< 99\%$ gemessen. Somit werden diese Optimierungen hier nicht weiter betrachtet.

Loop Deindexing

Das Diagramm 5.2 zeigt die detaillierten Ergebnisse durch die Anwendung der Optimierung *Loop Deindexing*. Die Messungen zeigen vergleichbare Werte für beide Architekturen, mit deutlichen Ausreißern bei den Benchmarks fir_31_1, ludcmp und selection_sort. Für diese Programme benötigt der TriCore-Prozessor signifikant mehr Zeit. Umgekehrt wird bei latnrm_32_64 eine deutliche Laufzeitzunahme auf der ARM-Architektur gemessen. Das beste Ergebnis mit 77,2% der relativen ACET wird für ARM bei selection_sort gemessen, das schlechteste mit 116,2% bei latnrm_32_64. Für TriCore sind das edge_detect mit 76,2% und fir_32_1 mit 125,4% der relativen ACET. Die durchschnittliche ACET-Veränderung beträgt 99,78% der relativen ACET für ARM und 105,46% für TriCore. Somit zeigt diese Optimierung für beide Architekturen keine signifikante Verbesserung der ACET.

Loop Head Transformation

Die Diagramme 5.3 und 5.4 zeigen die detaillierten Ergebnisse durch die Anwendung der Optimierung *Loop Head Transformation*. Da diese Optimierung auf alle Schleifen ange-

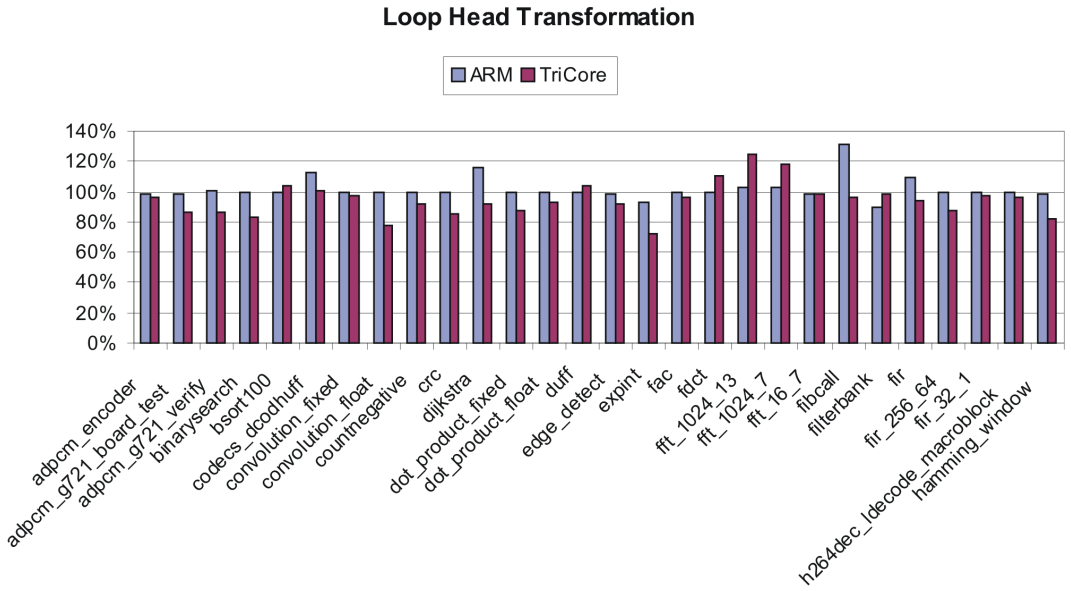


Abbildung 5.3.: Einfluss von Loop Head Transformation auf die ACET unterschiedlicher Architekturen

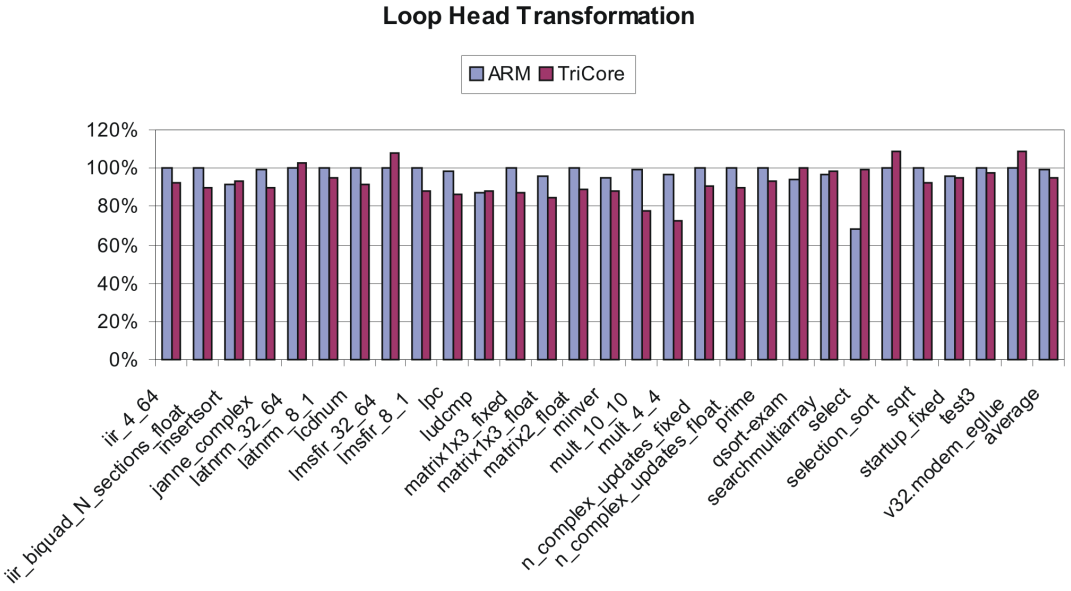


Abbildung 5.4.: Einfluss von Loop Head Transformation auf die ACET unterschiedlicher Architekturen

wendet werden kann, wurden wie erwartet fast alle Benchmarks optimiert. Für die meisten Programme kann eine deutliche Reduktion der ACET auf dem TriCore-Prozessor im Gegensatz zu ARM festgestellt werden. Ausnahmen bilden die Benchmarks fft_1024_13, fft_1024_7 und select, für die auf dem ARM eine bessere ACET gemessen wurde. Das beste Ergebnis für ARM mit 68,1% der relativen ACET wurde mit select erzielt, das schlechteste mit 131,6% für fibcall. Für TriCore liegt das beste Ergebniss bei expint mit 72,6% der relativen ACET, das schlechteste bei fft_1024_13 mit 124,8%. Im Durchschnitt ergibt sich eine relative ACET von 99,07% für die ARM-Plattform und 95,2% für die TriCore-Plattform. Somit kann durch diese Optimierung auf dem TriCore-Prozessor eine Verbesserung der ACET um bis zu 5% erreicht werden, auf dem ARM dagegen wird die Laufzeit kaum beinflusst.

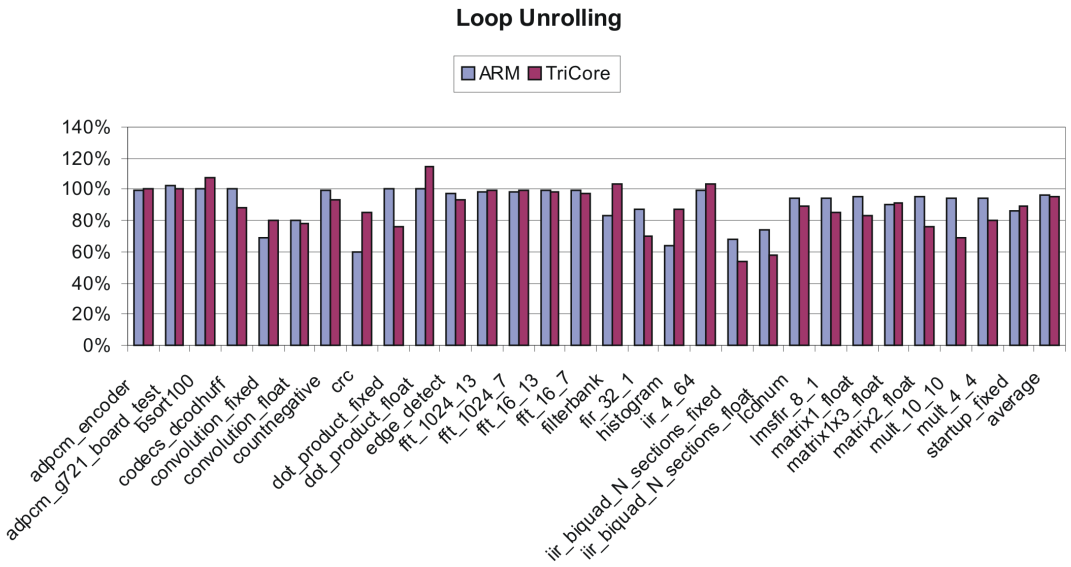


Abbildung 5.5.: Einfluss von Loop Unrolling auf die ACET unterschiedlicher Architekturen

Loop Unrolling

Das Diagramm 5.5 zeigt die detaillierten Ergebnisse durch die Anwendung der Optimierung *Loop Unrolling*. Die Auswirkungen dieser Optimierung hängen stark von der Größe der Schleifen im Programm ab. Somit variieren die Messergebnisse erwartungsgemäß sehr stark, so dass kein eindeutiger Vorteil für eine Architektur festgestellt werden kann. Das beste Ergebnis für die ARM-Plattform wurde bei crc mit 60,08% der relativen ACET gemessen, das schlechteste bei adpcm_g721_board_test mit 102,1%. Für die TriCore-Plattform sind das entsprechend iir_biquad_N_sections_fixed mit 54,06% und dot_product_float mit 115,1%. Die durchschnittliche relative ACET für alle Benchmarks beträgt 96,14% für ARM und 95,15% für TriCore. Diese Optimierung trägt also spürbar zur ACET-Minimierung bei

beiden Plattformen bei. Deshalb sollte diese Optimierung immer ausgeführt werden, da die nur in sehr seltenen Ausnahmefällen zur Verschlechterung der Laufzeit führt und die ACET fast immer verbessert.

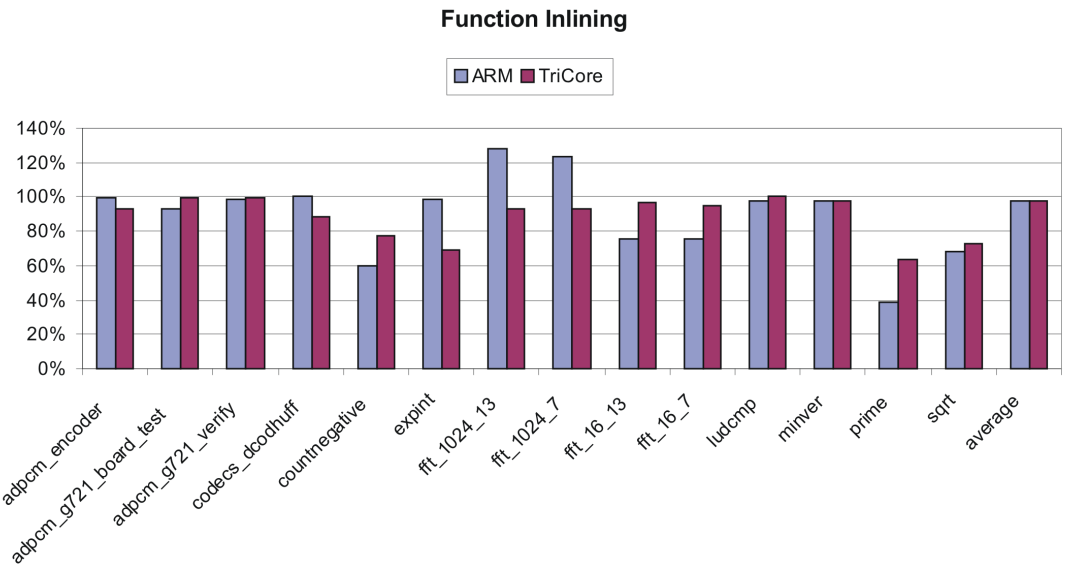


Abbildung 5.6.: Einfluss von Function Inlining auf die ACET unterschiedlicher Architekturen

Function Inlining

Das Diagramm 5.6 zeigt die detaillierten Ergebnisse durch die Anwendung der Optimierung *Function Inlining*. Ähnlich wie im *Loop Unrolling* hängt auch hier das Ergebnis der Messung stark von dem betrachteten Programm ab, was die erhebliche Varianz in den relativen ACET-Werten für die einzelne Benchmarks erklärt. Der architekturbedingte Unterschied lässt sich sehr gut an den Fast-Fourier-Transformation Benchmarks fft_1024_x und fft_16_x beobachten. Während die hohe Präzision der float-Werte bei fft_1024 (wo Fließkommazahlen mit 1024 Nachkommastellen benutzt werden) den ARM-Prozessor stark ausbremst, zeigt der TriCore aufgrund der integrierten FPU für die beiden Benchmarks in etwa die gleiche Laufzeit. Das beste Ergebnis beim Einsatz der *Function Inlining*-Optimierung wird für beide Architekturen bei prime gemessen, mit 38,30% der relativen ACET für ARM und 63,40% für TriCore. Das schlechteste Ergebnis liegt für ARM bei fft_1024_13 mit 128,2% der relativen ACET und für TriCore bei ludcmp mit 100,3%. Im Durchschnitt ergibt sich eine vergleichbare ACET von 97,91% für ARM und 97,77% für TriCore.

5.3. Vergleich der Einflüsse von Standardoptimierungen auf ACET und Energieverbrauch

Die im vorhergehenden Abschnitt untersuchten Optimierungen wurden zusätzlich auf deren Auswirkungen auf den Energieverbrauch untersucht. Da die Energiedatenbank nur die ARM-Architektur unterstützt, beschränken sich die folgenden Messungen auf den ARM7TDMI-Prozessor.

5.3.1. Vergleich der Standardoptimierungsstufen

Zuerst werden wieder die Verbesserungen der Standardoptimierungen mit Optimierungsstufen `O1` bis `O3` untersucht. Die Y-Achse stellt hier die relative ACET bzw. den Energieverbrauch dar. Weiterhin wird der Mittelwert der gemessenen relativen ACET über die einzelnen Benchmarks in jedem Benchmark-Suite dargestellt, mit Referenzwert von 100% als der entsprechende Mittelwert bei der Optimierungsstufe `O0`.

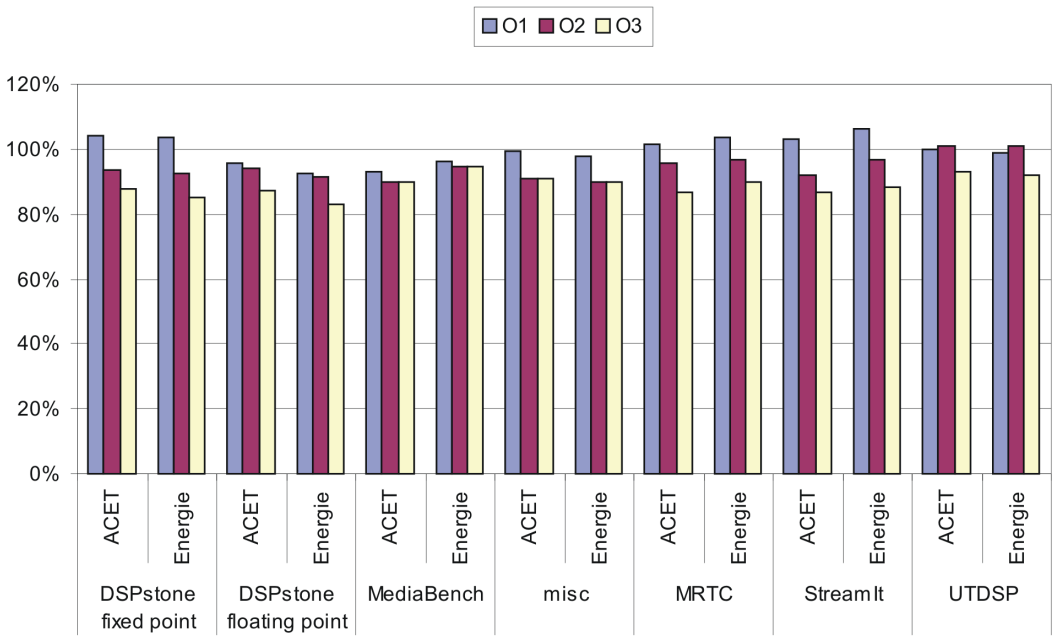


Abbildung 5.7.: Vergleich ACET- und Energieverbrauchreduktion bei unterschiedlichen Optimierungsstufen auf dem ARM7TDMI

Die Auswirkungen der Standardoptimierungen auf die beiden Kriterien sind in diesem Fall nahezu gleich. Eine Minimierung der ACET über die Standardoptimierungsstufen führt auch immer zu einer Reduktion des Energieverbrauchs des Programms. Umgekehrt bedeutet das, dass auch eine Reduktion des Energieverbrauchs die ACET verringert.

Der beste Messwert für die relative ACET wurde mit 86,9% bei der Benchmark-Suite MRTC mit der Optimierungsstufe `O3` ermitellt, der schlechteste mit 86,9% bei DSPstone fixed point mit `O1`. Für den Energieverbrauch wurde das beste Ergebnis mit 83,1% im Vergleich zu `O0` bei DSPstone floating point mit der Optimierungsstufe `O3` gemessen und das schlechteste mit 106,2% bei StreamIt bei `O1`. Im Durschnitt wird bei `O1` eine Wert von 99,6% für ACET und 99,9% für Energieverbrauch erreicht, bei `O2` entsprechend 93,9% für ACET und 94,6% für Energieverbrauch und bei `O3` 89% für ACET und 88,9% für Energieverbrauch.

5.3.2. Vergleich der einzelnen Optimierungen

Anschließend wird der Einfluss von einzelnen Optimierungen auf den Energieverbrauch untersucht. Wegen der gleichen Überlegungen wie bei den Messungen auf unterschiedlichen Archikturen wird bei den einzelnen Optimierungen als Referenz im Hinblick auf die ACET wieder die Optimierungsstufe `O1` verwendet. Auch hierbei wurden nur Benchmarks in die Diagramme aufgenommen, die eine messbare Änderung der ACET oder des Energieverbrauchs auweisen konnten.

Die Optimierungen *Loop Collapsing*, *Loop Unswitching* und *Struct Scalarization*, die schon bei vorangegangenen Messungen zu keiner Änderungen der ACET führten, wurden auch bei dieser Auswertung nicht beachtet. Für alle ausgeführten Benchmarks zeigte sich kein Einfluss auf den Energieverbrauch durch diese Optimierungen.

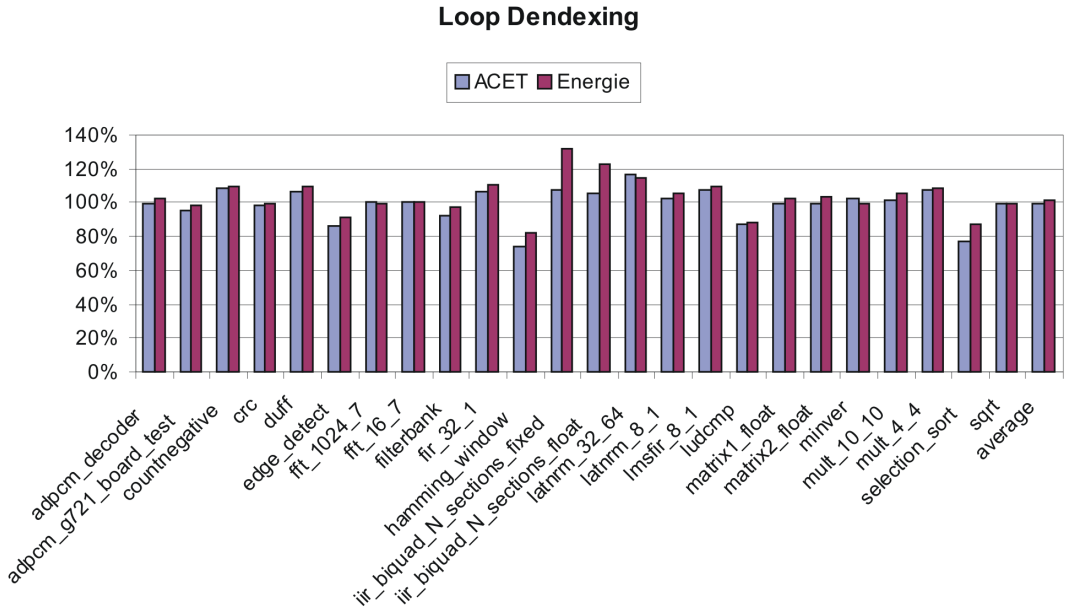


Abbildung 5.8.: Einfluss von Loop Deindexing auf die ACET und den Energieverbrauch

Loop Deindexing

Das Diagramm 5.8 zeigt die detaillierten Ergebnisse durch die Anwendung der Optimierung *Loop Deindexing*. Für die meisten Benchmarks besteht ein direkter Zusammenhang zwischen der resultierenden relativen ACET und Energieverbrauch. Dabei kann durch die Optimierung der Energieverbrauch auch vergrößert werden. Dies ist insbesondere an den Benchmarks iir_biquad_N_sections_fixed und iir_biquad_N_sections_float gut sichtbar. Hierbei wird der relative Energieverbrauch von 122,7% bzw. 132,3% gemessen. Der beste Wert für den Energieverbrauch wurde bei hamming_window mit 82,30% erzielt. Insgesamt ergibt sich die durchschnittliche relative ACET zu 99,78% des Referenzwertes, und der relative Energieverbrauch zu 101,16%. Somit ist diese Optimierung zur einen gleichzeitigen ACET- und Energieoptimierung nicht geeignet.

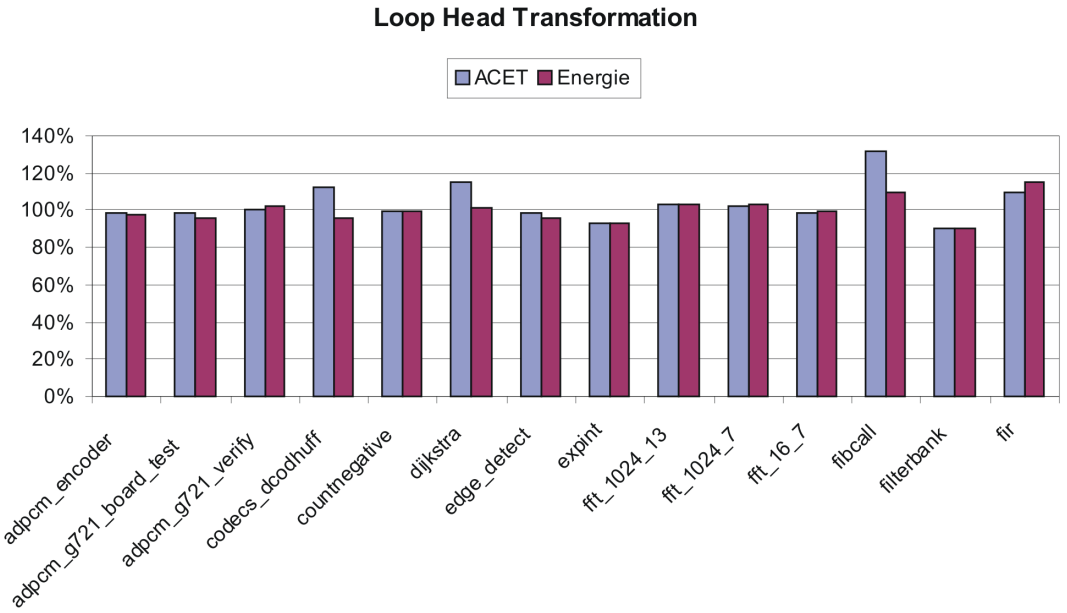


Abbildung 5.9.: Einfluss von Loop Head Transformation auf die ACET und den Energieverbrauch

Loop Head Transformation

Die Diagramme 5.9 und 5.10 zeigen die detaillierten Ergebnisse durch die Anwendung der Optimierung *Loop Head Transformation*. Auch hierbei hängt der Energieverbrauch von der ACET ab, dabei wird der Energieverbrauch durch die Optimierung meistens etwas stärker minimiert. Das beste Ergebnis für den Energieverbrauch wurde bei select gemessen, mit 70,80% im Vergleich zu `O1` und das schlechteste bei fir mit 114,70%. Im Durchschnitt verringert sich die relative ACET auf 99,07% und der Energieverbrauch auf 98,20%.

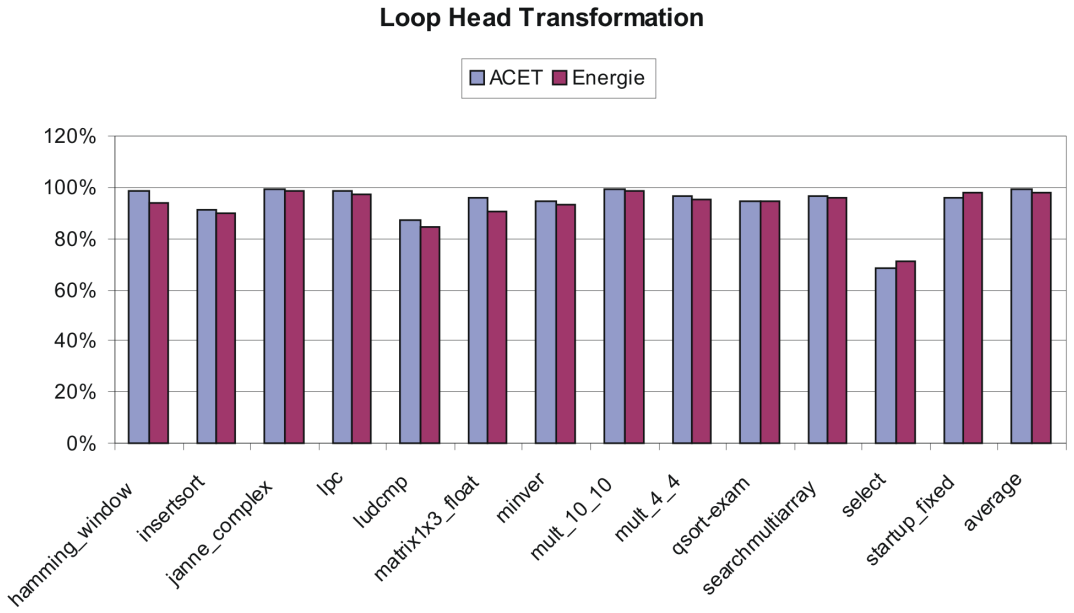


Abbildung 5.10.: Einfluss von Loop Head Transformation auf die ACET und den Energieverbrauch

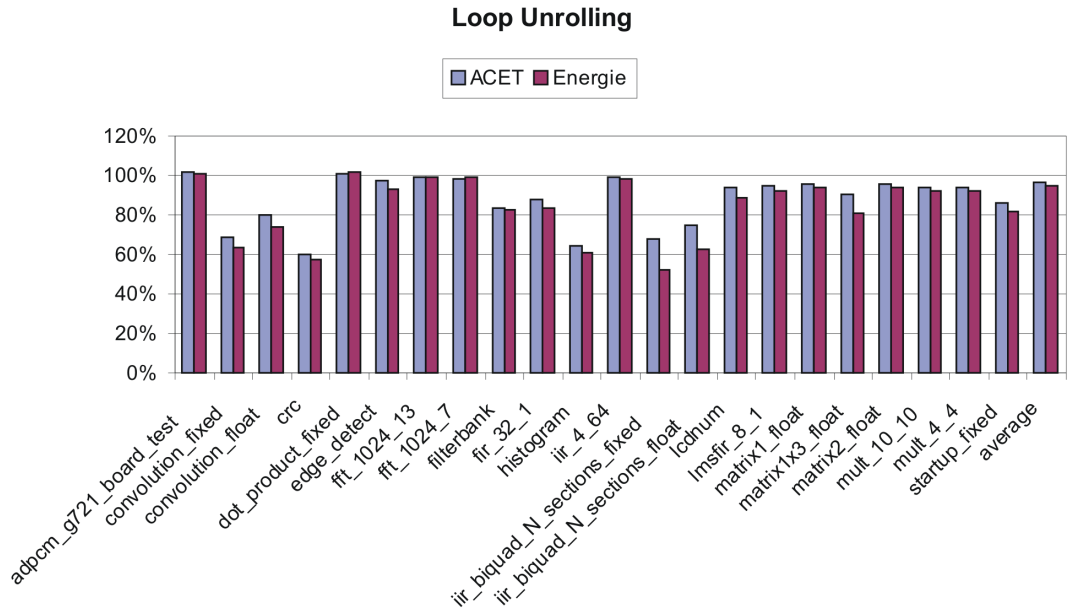


Abbildung 5.11.: Einfluss von Loop Unrolling auf die ACET und den Energieverbrauch

Loop Unrolling

Das Diagramm 5.11 zeigt die detaillierten Ergebnisse durch die Anwendung der Optimierung *Loop Unrolling*. Auch bei dieser Optimierung entwickelt sich der Energieverbrauch ähnlich wie die ACET. Auffallend ist, dass hierbei insbesondere die Benchmarks iir_biquad_N_sections_fixed und iir_biquad_N_sections_float sehr geringe Energieverbrauchswerte aufweisen. Der positive Einfluss auf den Energieverbrauch verhält sich hier gegensätzlich zu Optimierung *Loop Deindexing*. Das beste Ergebnis mit 52,11% des relativen Energieverbrauchs wurde bei iir_biquad_N_sections_fixed gemessen, das schlechteste mit 101,97% bei dot_product_fixed. Das beste Messwert für die relative ACET wurde bei crc mit 60,08% gemessen, das schlechteste bei adpcm_g721_board_test mit 102,1%. Die durchschnittlichen Werte für die relative ACET betragen 96,14% des Referenzwertes, und für den Energieverbrauch 94,97%. Im Gegensatz zu den vorher betrachteten Optimierungen kann somit mit Loop Unrolling der Energieverbrauch merklich verringert werden.

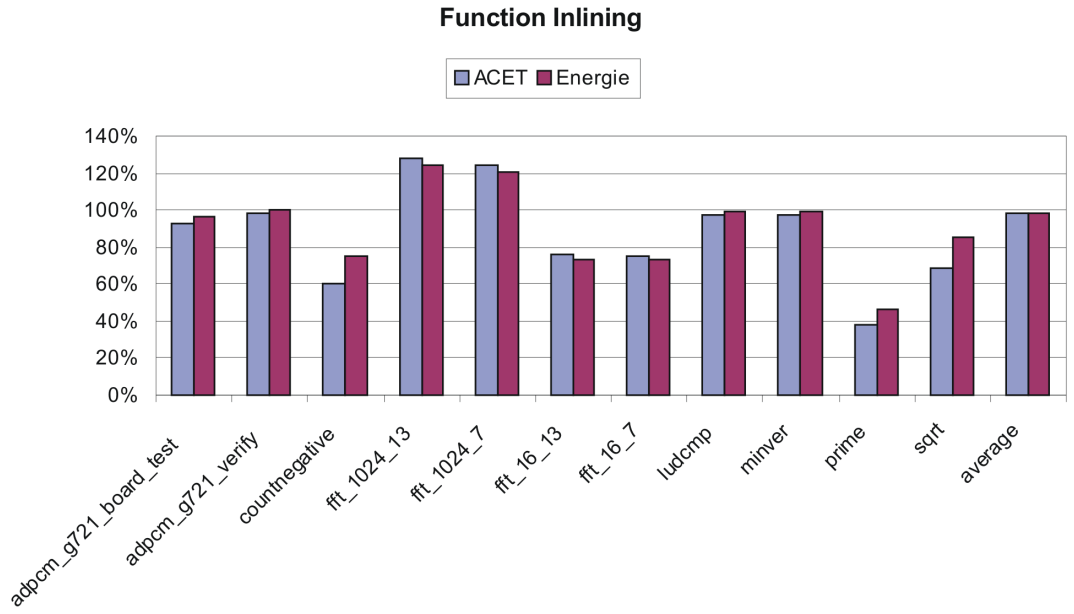


Abbildung 5.12.: Einfluss von Function Inlining auf die ACET und den Energieverbrauch

Function Inlining

Das Diagramm 5.12 zeigt die detaillierten Ergebnisse durch die Anwendung der Optimierung *Function Inlining*. Auch wenn der in anderen Optimierungen beobachtete Zusammenhang zwischen ACET und Energie auch hier sichtbar wird, wird durch *Function Inlining* die ACET meistens viel stärker reduziert. Offenbar wirkt sich der Overhead beim Funktionsaufruf nicht so stark auf den Energieverbrauch, wie die eigentliche Ausführung des

Funktionsrumpfes. Die schlechten Werte bei fft_1024_x sind durch die ARM-Architektur zu erklären, die Fließkommazahlen über Software berechnet. Das beste Ergebnis für den relativen Energieverbrauch wurde bei prime mit 46,80% erzielt und das schlechteste bei fft_1024_13 mit 123,8%. Für ACET liegt der beste Messwert mit 38,30% bei prime und der schlechteste mit 128,2% bei fft_1024_13. Im Durchschnitt ergibt sich die relative ACET von 97,91% und der relative Energieverbrauch von 98,5%.

5.4. Fazit

In diesem Kapitel wurden die Auswirkungen der Standardoptimierungen zum einem auf zwei unterschiedliche Architekturen und zum anderem auf die ACET und Energie verglichen. Für die Standardoptimierungen, die über die Optimierungsstufen `O1` bis `O3` im WCC eingeschaltet werden, entspricht das gemessene Verhalten dem von Optimierungen erwarteten: bei jeder höheren Optimierungsstufe wird die ACET auf den beiden Architekturen verringert. Der Energieverbrauch auf dem ARM7TDMI-Prozessor wird in ähnlich Größenordnungen wie die ACET reduziert.

Ein etwas anderes Bild ergibt sich bei der Betrachtung einzelner Optimierungen. So ergibt der Einsatz von Optimierungen *Loop Collapsing*, *Loop Unswitching* und *Struct Scalarization* keinen Unterschied bei den gemessenen Werten für alle Benchmarks. Der Einsatz von *Loop Deindexing* kann sogar zur Verschlechterung der untersuchten Kriterien auf beiden Architekturen führen. Alle anderen Optimierungen erwirken im Durchschnitt die Minimierung der Kriterien, sowohl für ACET auf dem ARM7TDMI und TC1797 als auch für ACET und Energieverbrauch auf dem ARM-Prozessor. Allerdings können die Ergebnisse für einzelne Programme sich stark variieren und sogar zur Verschlechterung im Vergleich zum nicht-optimierten Fall führen. Von daher sollten die Ergebnisse der Optimierungen anschließend immer überprüft werden. Der Feedback-directed Ansatz, der durch diese Diplomarbeit ermöglicht wurde ist von daher dem heuristischen klar vorzuziehen.

Insgesamt sollten die Optimierungen *Loop Unrolling* und *Function Inlining* immer ausgeführt werden, da hierbei die meisten Einsparungen sowohl für ACET auf den beiden Architekturen als auch für den Energieverbrauch auf der ARM-Plattform zu erwarten sind. Dagen sind Verschlechterungen der Kriterien nur selten.

6. Zusammenfassung / Ausblick

Die beiden Ziele dieser Diplomarbeit waren die Erweiterung des Compiler-Frameworks WCC um ein retargierbares Compiler-Backend und Entwicklung von Techniken zur Unterstützung von multikriteriellen Optimierungen.

Die wichtigsten Ergebnisse der Arbeit werden in Abschnitt 6.1 kurz vorgestellt. Anschließend wird in Abschnitt 6.2 ein Überblick über weitere ausstehende und mögliche Arbeiten gegeben.

6.1. Zusammenfassung

Einleitend wurden in Kapitel 1 die Ziele der Diplomarbeit motiviert, indem unterschiedliche Einsatzszenarien für retargierbare Compiler und multikriterielle Optimierungen aufgezeigt wurden. Kapitel 2 stellte dann mit dem Infineon TriCore und ARMv4 die Zielarchitekturen vor, die in dieser Arbeit im WCET-optimierenden Compiler-Framework WCC eingesetzt wurden. Nachfolgend wurden mit dem GCC, dem encc und dem WCC drei optimierende C-Compiler für eingebettete Systeme vorgestellt. Die Beschreibungen des internen Aufbaus der Frameworks sollten einerseits den Entwicklungsstand von modernen Compiler illustrieren und andererseits die Stellen aufzeigen, an denen die in dieser Arbeit entwickelten Erweiterungen des WCC ansetzen.

Vor Beginn der Diplomarbeit unterstütze das Compiler-Framework des WCC mit dem Infineon TriCore nur eine einzige Zielarchitektur. Das Kapitel 3 befasste sich mit der Aufgabe, das Compiler-Backend retargierbar zu machen. Dazu wurden zuerst die Grundlagen von retargierbaren Compiler eingeführt, um anschließend die Möglichkeiten zur Erweiterung des Frameworks aufzuzeigen. Ausgehend davon wurden die notwendigen Änderungen am WCC-Framework beschrieben, um die gewünschte Retargierbarkeit zu realisieren. Dazu wurde erstens der ursprüngliche Codeselector durch einen Zielcompiler ersetzt sowie die LLIR an die neue Zielarchitektur angepasst. Um den Assemblercode, der vom Zielcompiler generiert wird, in die LLIR zu übersetzen wurde ein spezielles Werkzeug, GCC2LLIR, entwickelt. Dieser Textparser übernimmt die Transformation des Assemblercodes in die interne Darstellung der LLIR. Die realisierten Erweiterungen wurden am Beispiel des ARM7TDMI-Prozessor implementiert.

Ein weiteres Ziel der Diplomarbeit war es, das Compiler-Framework des WCC um Unterstützung für multikriterielle Optimierungen zu erweitern. Das Kapitel 4 befasst sich mit dieser Aufgabe. Zuerst wurde die Thematik von multikriteriellen Optimierungen eingeführt und die dabei entstehende Herausforderungen beschrieben. Nachfolgend wurde auf die Darstellung einzelner Kriterien im Compiler eingegangen und die Kriterien näher vor-

gestellt. Des Weiteren wurden die Methoden zur Messung der einzelnen Krierien beschrieben. Beispielhaft wurde für die Bestimmung der ACET und des Energieverbrauchs eines zu übersetzenden Programms im Laufe dieser Diplomarbeit ein Profiling-Modul entwickelt und in das Compiler-Framework des WCC integriert. Dazu wurde ein Simulator für beide Zielarchitekturen eingebunden, der Laufzeit-Traces eines Programms erstellt. Durch Auswertung dieser Traces können zum einen Laufzeiten und Ausführungshäufigkeiten für Funktionen und Basisblöcke bestimmt werden. Zum anderen wurde durch die Anbindung einer Energiedatenbank ebenfalls für Funktionen und Basisblöcke eine Berechnung des Energieverbrauchs realisiert, die auch Speicher und Cache-Zugriffe berücksichtigt. Um die Ergebnisse im WCC für weitere Optimierungen verfügbar zu machen, wurden zwei neue Objective-Klassen von der Klasse LLIR_Objective abgeleitet. Damit werden die ACET- und Energieverbrauchswerte in der LLIR annotiert. Anhand dieser Informationen können somit im Compiler anschließend Feedback-gesteuerte Optimierungen angewendet werden. Die Implementierung des Moduls und das Interface der Objective-Klassen werden am Ende des Kapitels vorgestellt.

Abschließend wurden im Kapitel 5 die Ergebnisse von Standardoptimierungen auf unterschiedlichen Architekturen und für verschiedene Kriterien untersucht. Dabei wurden erstens die Auswirkungen auf die ACET beim Einsatz von Standardoptimierungen auf unterschiedlichen Architekturen verglichen. Zweitens wurde gemessen, inwiefern der Einsatz von Standartoptimierungen sich auf die gleichzeitige Minimierung von ACET und Energieverbrauch auf dem gleichen Prozessor auswirkt.

Der Einsatz von Standardoptimierungsstufen O0 bis O3 des WCC auf den beiden Architekturen ARM und TriCore führt zu den von solchen Optimierungen erwarteten Ergebnissen, mit jeder höheren Optimierungsstufe wird die ACET weiter verringert. Die Auswirkungen von einzelnen Optimierungen auf die jeweilige Architektur fällt allerdings sehr unterschiedlich aus. Insgesamt sollten immer die Optimierungen *Loop Unrolling* und *Function Inlining* ausgeführt werden. Durch diese zwei Optimierungen kann in den meisten Fällen eine nennenswerte Verringerung der ACET erreicht werden. Schlechtere Ergebnisse sind dagegen nur sehr selten zu erwarten.

Der Einsatz von Standardoptimierungsstufen O0 bis O3 des WCC und die Auswertung der Ergebnisse im Hinblick auf den Einfluss auf die ACET und den Energieverbrauch eines Programms führt zu der Schlussfolgerung, dass die ACET und der Energieverbrauch direkt voneinenader abhängen. Wenn eins von diesen Kriterien minimiert wird, führt das zu einer Reduktion des anderen in ähnlicher Größenordnung. Auch hierbei sollten immer die beiden Optimierungen *Loop Unrolling* und *Function Inlining* angewendet werden, da sie in den meisten Fällen eine signifikante Reduktion sowohl der ACET als auch des Energieverbrauchs erwirken.

6.2. Ausblick

Auch wenn die grundlegenden Mechanismen zur Erweiterung des Compiler-Frameworks erfolgreich umgesetzt wurden, so gibt es eine Reihe von Ansatzpunkten, die als Ausgang

für weitere Arbeiten dienen können.

- Die statische WCET-Analyse ist im Moment nur für den Infineon TriCore vorhanden. Diese Analyse sollte auch für das implementierte Compiler-Backend für die ARM-Architektur zur Verfügung stehen. Dazu müsste der LLIR2CRL-Converter sowie der CRL2LLIR-Converter für das Analysetool aiT an den ARM7TDMI-Prozessor angepasst werden.

- Unter dem Hamming-Abstand von zwei Wörtern auf einem Bus wird die Anzahl der unterschiedlichen Bits der beiden Wörter verstanden. Um ein neues Wort auf den Bus zu legen, müssen die Leitungen, auf denen die vom letzen Wort unterscheidende Bits angelegt sind, von 0 auf 1 oder umgekehrt umgeladen werden. Dabei wird Energie verbraucht, die direkt von dem Hamming-Abstand der beiden Wörter abhängt. Die Betrachtung der Hamming-Abstände und der damit zusammenhängenden Kosten für den Energieverbrauch eines Programms können das bereits integrierte Energiemodell noch mehr verbessern. Da die unterschiedliche Werte auf den Bussen bis zu 13% des Gesamtenergieverbrauchs ausmachen [Kna01], sollte ein Mechanismus zur Berechnung der Hamming-Kosten in LLIRProfile integriert werden.

- In Zukunft sollten Optimierungen entwickelt werden, die für unterschiedliche Architekturen den Energieverbrauch minimieren. Dazu muss, wie in Kapitel 4.2.3 beschrieben, für jede unterstützte Architektur ein Energiemodell entwickelt werden, um den Energieverbrauch eines Programms zu analysieren. Momentan existiert nur ein genaues Energiemodell für den Thumb-Befehlssatz der ARMv4-Architektur. Für alle weiteren Architekturen, also insbesondere für den Infineon TriCore müsste dieses Energiemodell neu erstellt werden.

- In der Diplomarbeit von Fatih Gedikli [Ged08] wird die Transformation von WCET-Informationen für die Nutzung in High-Level Optimierungen beschrieben. Ein ähnlicher Mechanismus kann auch für die Transformation von anderen Informationen angewendet werden. Die im WCC vorhandene Backannotation müsste um die Unterstützung von weiteren Kriterien erweitert werden, damit die Trace-Analysen aus der LLIR für die High-Level Optimierungen verfügbar werden. Durch den Einsatz eines Zielcompilers für das retargierbare Backend wird die Zuordnung der Funktionen und Basisblöcke aus der LLIR zu den entsprechenden Konstrukten der Quellsprache erschwert. Für die Zuordnung könnten z.B. vom Zielcompiler erzeugte DWARF Debuginformationen [dwa10] benutzt werden.

- Schließlich wird durch die Ergebnisse dieser Arbeit die Möglichkeit geschaffen, kriterien- oder plattformübergreifende Feedback-directed Optimierungen zu entwickeln und zu bewerten. Zum einen sind Ansätze vorstellbar, die eine Pareto-optimale Menge von Standardoptimierungen bestimmen. Dazu müssen die Auswirkungen von Optimierungen auf die gewünschten Kriterien gemessen und nur solche Optimierungen ausgewählt werden, die zum optimalen Gesamtergebnis für alle Kriterien führen. Zum

anderen können ganz neue Ansätze untersucht werden, wo die gleichzeitige Optimierung von mehreren Kriterien zum Ziel wird.

A. Verwendete Benchmarks

Die folgenden Benchmarks wurden in dieser Diplomarbeit verwendet:

DSPStone fixed point	DSPStone floating point	MRTC	UTDSP
adpcm_g721_board_test	complex_multiply_float	adpcm_decoder	compress
adpcm_g721_verify	complex_update_float	adpcm_encoder	edge_detect
complex_multiply_fixed	convolution_float	binarysearch	fir_32_1
complex_update_fixed	dot_product_float	bsort100	fir_256_64
convolution_fixed	fir_float	countnegative	histogram
dot_product_fixed	fir2dim_float	crc	iir_1_1
fft_16_7	iir_biquad_N_sections_float	duff sqrt	iir_4_64
fft_16_13	lms_float	qsort-exam	jpeg
fft_1024_7	matrix1_float	expint	latnrm_8_1
fft_1024_13	matrix1x3_float	fac fdct	latnrm_32_64
fir_fixed	matrix2_float	select	lmsfir_8_1
fir2dim_fixed	n_complex_updates_float	fibcall	lmsfir_32_64
iir_biquad_N_sections_fixed	n_real_updates_float	fir	lpc
lms_fixed	real_update_float	insertsort	mult_4_4
matrix1x3_fixed		janne_complex	mult_10_10
n_complex_updates_fixed		lcdnum	qmf_receive
n_real_updates_fixed		ludcmp	qmf_transmit
real_update_fixed		matmult	spectral
startup_fixed		minver	v32.modem_eglue
statemate		petrinet	v32.modem_cnoise
		prime	

MediaBench	MiBench	misc	StreamIt
h264dec_ldecode_macroblock	dijkstra	codecs_dcodhuff	filterbank
		hamming_window	
		searchmultiarray	
		selection_sort	
		test3	

Tabelle A.1.: Innerhalb der Diplomarbeit verwendete Benchmarks

B. Messergebnisse

		O1	O2	O3
DSPstone fixed point	ARM ACET	104,2%	93,7%	88,0%
	TriCore ACET	98,0%	92,4%	83,6%
	ARM Energie	103,5%	92,5%	84,9%
DSPstone floating point	ARM ACET	95,6%	94,0%	87,4%
	TriCore ACET	88,4%	80,8%	74,2%
	ARM Energie	92,7%	91,3%	83,1%
MediaBench	ARM ACET	93,2%	89,6%	89,7%
	TriCore ACET	98,0%	94,3%	94,3%
	ARM Energie	96,4%	94,5%	94,5%
misc	ARM ACET	99,3%	91,1%	91,0%
	TriCore ACET	94,7%	106,2%	99,8%
	ARM Energie	97,8%	89,6%	89,6%
MRTC	ARM ACET	101,5%	95,6%	86,9%
	TriCore ACET	94,8%	88,3%	87,6%
	ARM Energie	103,7%	96,9%	89,7%
StreamIt	ARM ACET	103,2%	92,1%	86,5%
	TriCore ACET	88,0%	77,1%	79,5%
	ARM Energie	106,2%	96,5%	88,5%
UTDSP	ARM ACET	99,9%	100,8%	93,1%
	TriCore ACET	94,4%	106,5%	96,1%
	ARM Energie	98,7%	100,7%	91,9%

Tabelle B.1.: Messergebnisse für die Standartoptimierungsstufen O1-O3

	ARM ACET	TriCore ACET	ARM Energie
adpcm_decoder	99,50%	100%	102,50%
adpcm_g721_board_test	95,20%	99,70%	98,70%
bsort100	99,70%	105,80%	99,70%
countnegative	108,30%	94,20%	109,60%
crc	98,30%	93,00%	99,30%
duff	107,00%	104,30%	110,00%
edge_detect	86,00%	76,20%	90,90%
fft_1024_7	100,30%	101,20%	99,80%
fft_16_7	100,30%	98,00%	100,30%
filterbank	92,60%	94,50%	97,70%
fir_256_64	100,00%	102,50%	100,00%
fir_32_1	107,00%	125,40%	110,30%
hamming_window	73,80%	172,10%	82,30%
histogram	99,20%	97,40%	99,20%
iir_4_64	100,50%	95,70%	100,50%
iir_biquad_N_sections_fixed	107,80%	109,30%	132,30%
iir_biquad_N_sections_float	105,70%	115,70%	122,70%
latnrm_32_64	116,20%	86,50%	114,70%
latnrm_8_1	102,60%	144,00%	105,40%
lmsfir_32_64	100,30%	95,80%	100,00%
lmsfir_8_1	107,10%	209,80%	109,60%
ludcmp	87,70%	118,90%	88,40%
matrix1_float	99,90%	106,80%	102,90%
matrix2_float	99,90%	104,20%	103,00%
minver	102,80%	102,30%	99,90%
mult_10_10	101,20%	177,00%	105,50%
mult_4_4	108,00%	159,10%	108,90%
selection_sort	77,20%	105,50%	87,70%
sqrt	99,70%	99,00%	99,70%
average	99,78%	105,46%	101,16%

Tabelle B.2.: Messergebnisse für Loop Deindexing

	ARM ACET	TriCore ACET	ARM Energie
adpcm_encoder	99,51%	99,98%	99,50%
adpcm_g721_board_test	102,10%	100,17%	100,90%
bsort100	99,94%	107,24%	100,00%
codecs_dcodhuff	100,00%	87,83%	100,00%
convolution_fixed	68,55%	80,23%	63,84%
convolution_float	79,81%	78,21%	74,33%
countnegative	99,88%	93,31%	99,88%
crc	60,08%	85,12%	57,79%
dot_product_fixed	100,72%	76,47%	101,97%
dot_product_float	100,00%	115,14%	100,00%
edge_detect	97,09%	92,83%	93,23%
fft_1024_13	98,90%	99,84%	99,26%
fft_1024_7	98,58%	99,81%	98,93%
fft_16_13	99,40%	98,79%	100,00%
fft_16_7	99,23%	97,32%	100,00%
filterbank	83,63%	103,97%	82,32%
fir_32_1	87,71%	69,87%	83,66%
histogram	64,00%	87,57%	61,00%
iir_4_64	99,00%	103,98%	98,45%
iir_biquad_N_sections_fixed	68,23%	54,06%	52,11%
iir_biquad_N_sections_float	74,35%	57,46%	62,25%
lcdnum	94,34%	89,51%	88,36%
lmsfir_8_1	94,84%	85,71%	92,61%
matrix1_float	95,66%	83,50%	93,60%
matrix1x3_float	90,54%	91,70%	81,20%
matrix2_float	95,77%	75,76%	93,56%
mult_10_10	94,28%	68,99%	92,31%
mult_4_4	93,94%	80,15%	91,78%
startup_fixed	85,96%	88,88%	81,66%
average	96,14%	95,13%	94,97%

Tabelle B.3.: Messergebnisse für Loop Unrolling

	ARM ACET	TriCore ACET	ARM Energie
adpcm_encoder	99,90%	92,60%	99,90%
adpcm_g721_board_test	92,60%	99,30%	96,30%
adpcm_g721_verify	98,40%	99,30%	99,80%
codecs_dcodhuff	100,00%	88,70%	100,00%
countnegative	60,10%	77,40%	74,80%
expint	99,00%	68,90%	99,00%
fft_1024_13	128,20%	93,40%	123,80%
fft_1024_7	123,80%	93,10%	120,30%
fft_16_13	75,90%	96,30%	73,20%
fft_16_7	75,40%	94,60%	73,00%
ludcmp	97,50%	100,30%	99,00%
minver	97,70%	97,60%	99,60%
prime	38,30%	63,40%	46,80%
sqrt	68,50%	72,70%	85,50%
average	97,91%	97,77%	98,46%

Tabelle B.4.: Messergebnisse Function Inling

	ARM ACET	TriCore ACET	ARM Energie
adpcm_encoder	98,90%	96,10%	97,20%
adpcm_g721_board_test	98,70%	86,60%	96,10%
adpcm_g721_verify	100,10%	86,70%	102,50%
binarysearch	100,00%	82,80%	100%
bsort100	100,00%	103,40%	100%
codecs_dcodhuff	112,50%	100,90%	95,60%
convolution_fixed	100,00%	97,10%	100,00%
convolution_float	100,00%	77,70%	100,00%
countnegative	99,70%	91,70%	99,40%
crc	100,00%	85,70%	100,00%
dijkstra	115,40%	91,80%	101,60%
dot_product_fixed	100,00%	87,50%	100,00%
dot_product_float	100,00%	92,70%	100,00%
duff	99,90%	103,70%	99,90%
edge_detect	98,70%	92,30%	95,40%
expint	93,30%	72,60%	93,40%
fac	99,60%	96,10%	99,60%
fdct	100,00%	110,80%	100,00%
fft_1024_13	103,00%	124,80%	103,60%
fft_1024_7	102,60%	118,00%	103,30%
fft_16_7	98,50%	98,70%	99,20%
fibcall	131,60%	96,10%	109,20%
filterbank	90,20%	98,00%	90,30%
fir	109,60%	93,60%	114,70%
fir_256_64	100,00%	87,30%	100,00%
fir_32_1	100,00%	97,60%	100,00%
h264dec_ldecode_macroblock	99,80%	96,50%	100,00%
hamming_window	98,50%	82,00%	93,70%
iir_4_64	100,00%	92,80%	100,00%
iir_biquad_N_sections_float	100,00%	89,90%	100,00%
insertsort	91,20%	92,90%	90,00%
janne_complex	98,90%	90,20%	98,80%
latnrm_32_64	100,00%	102,40%	100,00%
latnrm_8_1	100,00%	94,60%	100,00%
lcdnum	100,00%	91,30%	100,00%
lmsfir_32_64	100,00%	107,90%	100,00%
lmsfir_8_1	100,00%	87,80%	100,00%

Tabelle B.5.: Messergebnisse für Loop Head Transformation

	ARM ACET	TriCore ACET	ARM Energie
lpc	98,50%	86,50%	97,40%
ludcmp	87,00%	88,30%	84,70%
matrix1x3_fixed	100,00%	87,60%	100,00%
matrix1x3_float	95,80%	84,70%	90,30%
matrix2_float	99,80%	89,10%	99,80%
minver	94,60%	88,10%	93,30%
mult_10_10	99,20%	77,60%	98,70%
mult_4_4	96,80%	72,30%	95,40%
n_complex_updates_fixed	100,00%	90,90%	100,00%
n_complex_updates_float	100,00%	90,00%	100,00%
prime	100,00%	93,60%	100,00%
qsort-exam	94,40%	100,00%	94,60%
searchmultiarray	96,50%	98,10%	95,80%
select	68,10%	98,90%	70,80%
selection_sort	100,00%	108,90%	100,00%
sqrt	100,00%	92,70%	100,00%
startup_fixed	95,90%	95,30%	97,80%
test3	100,00%	97,40%	100,00%
v32.modem_bencode	65,80%	99,50%	71,10%
v32.modem_eglue	100,00%	108,40%	100,00%
average	99,07%	95,20%	98,20%

Tabelle B.6.: Messergebnisse für Loop Head Transformation

Abbildungsverzeichnis

2.1. Blockdiagramm des TC1796-Prozessors 14
2.2. Blockdiagramm des ARM7TDMI-Prozessors 16
2.3. Kompilierung eines C-Programms mit GCC 20
2.4. GCC Frontend, Middleend und Backend 21
2.5. Arbeitsablauf des encc 23
2.6. Der Aufbau des WCET-aware C Compilers 25
2.7. Klassenhierarchie der ICD-LLIR 27
2.8. Arbeitsschritte bei der WCET-Berechnung mit aiT 30

3.1. Ersatz des internen Codeselectors durch einen Zielcompiler 36

4.1. Zusammenhang zwischen relativen WCET und Codegröße 44
4.2. Pareto-optimale Front für die Kriterien f_1 und f_2 46
4.3. Weighted Sum [Bur05] 48
4.4. ϵ - Constraint [Bur05] 49
4.5. NSGA II [Bur05] 51
4.6. Beispiel für einen Kontrollflussgraphen 54
4.7. Die Funktionen getACCallFrequency() und getACEC() 59

5.1. Vergleich ACET-Reduktion bei unterschiedlichen Optimierungsstufen und Architekturen 68
5.2. Einfluss von Loop Deindexing auf die ACET unterschiedlicher Architekturen 69
5.3. Einfluss von Loop Head Transformation auf die ACET unterschiedlicher Architekturen 70
5.4. Einfluss von Loop Head Transformation auf die ACET unterschiedlicher Architekturen 70
5.5. Einfluss von Loop Unrolling auf die ACET unterschiedlicher Architekturen 71
5.6. Einfluss von Function Inlining auf die ACET unterschiedlicher Architekturen 72
5.7. Vergleich ACET- und Energieverbrauchreduktion bei unterschiedlichen Optimierungsstufen auf dem ARM7TDMI 73
5.8. Einfluss von Loop Deindexing auf die ACET und den Energieverbrauch . . 74
5.9. Einfluss von Loop Head Transformation auf die ACET und den Energieverbrauch 75
5.10. Einfluss von Loop Head Transformation auf die ACET und den Energieverbrauch 76
5.11. Einfluss von Loop Unrolling auf die ACET und den Energieverbrauch . . . 76

5.12. Einfluss von Function Inlining auf die ACET und den Energieverbrauch . . 77

Literaturverzeichnis

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

[ARM04] ARM Limited: *ARM7TDMI Technical Reference Manual*. Revision r4p1, 2004

[ARM05] ARM Limited: *ARM Architecture Reference Manual*, 2005

[Bur05] Burke, Edmund (Hrsg.): *Search Methodologies : Introductory Tutorials in Optimization and Decision Support Techniques*. New York : Springer, 2005

[BW01] Burns, Alan ; Wellings, Andy: *Real-Time Systems and Programming Languages: ADA 95, Real-Time Java, and Real-Time POSIX*. Addison-Wesley, 2001

[che10] *Chess - retargetable C compiler*. http://www.retarget.com/, 2010. – Target Compiler Technologies

[cos10] *CoSy compiler development system*. http://www.ace.nl/, 2010. – ACE Associated Computer Experts

[CP01] Colin, Antoine ; Puaut, Isabelle: A Modular & Retargetable Framework for Tree-Based WCET Analysis. In: *Proc. of ECRTS*. Washington, DC, USA, June 2001

[Deb01] Deb, Kalyanmoy: *Multi-objective Optimization using Evolutionary Algorithms*. Chichester, UK : John Wiley & Sons, 2001

[dwa10] *The DWARF Debugging Standard*. http://dwarfstd.org, 2010

[EF+07] Elsner, Dean ; Fenlason, Jay u. a.: *Using as - The GNU Assembler, Version 2.19.1*, 2007. – Free Software Foundation, Inc.

[Ehr00] Ehrgott, Matthias: *Multicriteria Optimization*. Berlin : Springer, 2000

[EP07] Eckart, Jörg ; Pyka, Robert: *ICD Low Level Intermediate Representation Backend Infrastructure (LLIR) Developer Manual*. 2007. – Informatik Centrum Dortmund

[EP10] Eckart, Jörg ; Pyka, Robert: *ICD-LLIR Low-Level Intermediate Representation*. http://www.icd.de/es/icd-llir, 2010. – Informatik Centrum Dortmund

[FLT06] Falk, Heiko ; Lokuciejewski, Paul ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: *Proc. of ESTIMedia*. Seoul, Korea, 2006

[Ged08] Gedikli, Fatih: *Transformation und Ausnutzung von WCET-Informationen für High-Level-Optimierungen*, Technische Universität Dortmund, Diplomarbeit, 2008

[gnu10] *Die GNU-Projekt Homepage*. http://www.gnu.org/, 2010. – Free Software Foundation, Inc.

[GRE+01] Guthaus, Matthew R. ; Ringenberg, Jeffrey S. ; Ernst, Dan ; Austin, Todd M. ; Mudge, Trevor ; Brown, Richard B.: MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In: *Proc. of IISWC*. Austin, USA, 2001

[HP08] Hardy, Damien ; Puaut, Isabelle: Predictable Code and Data Paging for Real-Time Systems. In: *Proc. of ECRTS*. Prag, Tschechien, July 2008

[icd10] *ICD-CG code generator generator*. http://www.icd.de/es/icd-c/, 2010. – Informatik Centrum Dortmund

[Inf07] Infineon Technologies AG: *TC1796 User's Manual, v2.0*. München, 2007

[Inf09] Infineon Technologies AG: *TC1797 User's Manual, v1.1*. München, 2009

[JKK+05] Jaeger, Till ; Koglin, Olaf ; Kreutzer, Till ; Metzger, Axel ; Schulz, Carsten: *Die GPL kommentiert und erklärt : rechtliche Erläuterungen zur GNU general public license*. Köln : O'Reilly, 2005

[Kel07] Keller, John: Developers of real-time embedded software take aim at code complexity. In: *Military & Aerospace Electronics* (2007), April

[Kna01] Knauer, Markus: *Codierungsverfahren zur Reduktion des Energiebedarfs von Programmen*, Technische Universität Dortmund, Diplomarbeit, 2001

[Knu73] Knuth, Donald E.: *The art of computer programming*. Reading, MA : Addison-Wesley, 1973

[LCFM09] Lokuciejewski, Paul ; Cordes, Daniel ; Falk, Heiko ; Marwedel, Peter: A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models. In: *Proc. of CGO*. Seattle, USA, 2009

[LEM01] Lee, Sheayun ; Ermedahl, Andreas ; Min, Sang L.: An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors. In: *Proc. of LCTES*. New York, USA, 2001

[Leu97] Leupers, Rainer: *Retargetable Code Generation for Digital Signal Processors*. Boston : Kluwer Academic Publishers, 1997

[Leu00] Leupers, Rainer: *Code Optimization Techniqes for Embedded Processors - Methods, Algorithms and Tools.* Kluwer Academic Publishers, 2000

[LGM09] Lokuciejewski, Paul ; Gedikli, Fatih ; Marwedel, Peter: Accelerating WCET-driven Optimizations by the Invariant Path Paradigm: a Case Study of Loop Unswitching. In: *Proc. of SCOPES.* New York, USA, 2009

[LM01] Leupers, Rainer ; Marwedel, Peter: *Retargetable compiler technology for embedded systems : tools and applications.* Boston : Kluwer Academic Publishers, 2001

[Lok07] Lokuciejewski, Paul: *A WCET-Aware Compiler. Design, Concepts and Realization.* Vdm Verlag Dr. Müller, 2007

[LPMS97] Lee, Chunho ; Potkonjak, Miodrag ; Mangione-Smith, William H.: MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In: *Proc. of MICRO 30.* Research Triangle Park, NC, USA, 1997

[Mar95] Marwedel, Peter (Hrsg.): *Code generation for embedded processors.* Boston : Kluwer Academic Publishers, 1995

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

[Mäl10] Mälardalen WCET research group: *Mälardalen WCET benchmark suite.* http://www.mrtc.mdh.se/projects/wcet, 2010

[Nov04] Novillo, Diego: From Source to Binary: The Inner Workings of GCC. In: *Red Hat magazine* (2004), Dezember, Nr. 2

[PE10] Pyka, Robert ; Eckart, Jörg: *ICD-C Compiler Framework.* http://www.icd.de/es/icd-c, 2010. – Informatik Centrum Dortmund

[PLM08] Plazar, Sascha ; Lokuciejewski, Paul ; Marwedel, Peter: *A Retargetable Framework for Multi-objective WCET-aware High-level Compiler Optimizations.* Barcelona, Spanien, 2008

[SKWM01] Steinke, Stefan ; Knauer, Markus ; Wehmeyer, Lars ; Marwedel, Peter: An Accurate and Fine Grain Instruction-Level Energy Model Supporting Software Optimizations. In: *Proc. of PATMOS.* Yverdon-lesbains, Schweiz, 2001

[SS03] Srikant, Y. N. (Hrsg.) ; Shankar, Priti (Hrsg.): *The Compiler Design Handbook: Optimizations and Machine Code Generation.* Boca Raton : CRC Press, 2003

[Sta08] Stallman, Richard M.: *Using the GNU Compiler Collection (For gcc version 4.3.3)*, 2008

[Str10] StreamIt Entwickler: *StreamIt Testbench*. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml, 2010

[SWLM02] Steinke, Stefan ; Wehmeyer, Lars ; Lee, Bo-Sik ; Marwedel, Peter: Assigning Program and Data Objects to Scratchpad for Energy Reduction. In: *Proc. of DATE*. Paris, France, 2002

[SWV10] Steinke, Stefan ; Wehmeyer, Lars ; Verma, Manish: *Die encc Compiler Homepage*. http://ls12-www.cs.tu-dortmund.de/research/activities/encc, 2010

[sym10] *Die SymTA/S Homepage*. http://www.symtavision.com/, 2010. – Symtavision

[TMW94] Tiwari, Vivek ; Malik, Sharad ; Wolfe, Andrew: Power analysis of embedded software: a first step towards software power minimization. In: *Proc. of ICCAD*. Los Alamitos, USA, 1994

[UTD10] *UTDSP Benchmark Suite*. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html, 2010

[vas10] *Die VaST Systems Homepage*. http://www.vastsystems.com/, 2010

[vM94] Živojnović, C. S. J. Martinez M. J. Martinez ; Meyr, H.: DSPstone: A DSP-Oriented Benchmarking Methodology. In: *Proc. of ICSPAT*. Dallas, USA, 1994

[Weg05] Wegener, Ingo: *Theoretische Informatik : eine algorithmenorientierte Einführung*. Wiesbaden : Teubner, 2005

[WHM04] Wehmeyer, Lars ; Helmig, Urs ; Marwedel, Peter: Compiler-optimized Usage of Partitioned Memories. In: *Proc. of WMPI*. München, Deutschland, 2004

[WJ96] Wilton, Steven J. E. ; Jouppi, Norman P.: CACTI: An Enhanced Cache Access and Cycle Time Model. In: *IEEE Journal of Solid-State Circuits* 31 (1996)

[ZKW+04] Zhao, Wankang ; Kulkarni, Prasad ; Whalley, David u. a.: Tuning the WCET of Embedded Applications. In: *Proc. of RTAS*. Toronto, Canada, 2004

[ZWHM04] Zhao, Wankang ; Whalley, David ; Healy, Christopher ; Mueller, Frank: WCET Code Positioning. In: *Proc. of RTSS*. Lisbon, Portugal, 2004