UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Diplomarbeit

Untersuchung des Einflusses von Compiler-Optimierungen auf die maximale Programm-Laufzeit (WCET)

Martin Schwarzer

8. Januar 2007

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl XII
Fachbereich Informatik
Universität Dortmund

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

GERMANY · D-44221 DORTMUND

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Heutzutage halten Computersysteme Einzug in immer mehr Bereiche unseres täglichen Lebens. Personal Computer, die heute millionenfach in Büros und immer mehr privaten Haushalten für die verschiedensten Zwecke eingesetzt werden, spielen dabei allerdings nur eine untergeordnete Rolle. Ein sehr viel größeres Wachstum wird im Bereich der sogenannten eingebetteten Systeme erwartet. Eingebettete Systeme sind informationsverarbeitende Systeme, die in größere Produkte eingebettet und für den Benutzer meist nicht direkt sichtbar sind [Mar06]. Schon heute sind sehr viel mehr Prozessoren in eingebetteten Systemen im Einsatz als in herkömmlichen Computern. Die Computersysteme rücken also weiter in den Hintergrund, sind aber durch die Integration in immer mehr Produkte aus den verschiedensten Anwendungsgebieten allgegenwärtig. Deshalb wird diese Art der Informationsverarbeitung auch als *Ubiquitous Computing* bezeichnet.

Einige Beispiele für Anwendungsgebiete und Produkte, in denen eingebettete Systeme eingesetzt werden, sind die folgenden:

- Fahrzeuge: Airbag, ABS, ESP, Fly-by-Wire, Zugsteuerung
- Telekommunikation: ISDN, Mobiltelefone, PDAs
- Multimedia Geräte: DVD-Player, MP3-Player, Spielekonsolen
- Medizinische Geräte: Herzschrittmacher, Insulinpumpen
- Haushaltsgeräte: Waschmaschinen, Kühlschränke
- Industriesteuerungen: Robotersteuerung

Im Gegensatz zu herkömmlichen universell einsetzbaren Rechnern, mit denen der Benutzer beliebige Anwendungssoftware ausführen kann, werden eingebettete Systeme üblicherweise für eine ganz bestimmte Anwendung entwickelt. Dabei wird sowohl die Software als auch die Hardware speziell an die jeweilige Anwendung angepasst. Diese Spezialisierung ist auch notwendig, um den besonderen Anforderungen der Anwendungsgebiete, beispielweise nach geringem Energieverbrauch, Sicherheit, Zuverlässigkeit oder geringen Kosten, gerecht zu werden.

Häufig sind eingebettete Systeme auch Echtzeitsysteme, d.h. es müssen Zeitschranken bei der Programmausführung eingehalten werden. Die Korrektheit solcher Systeme hängt neben der Richtigkeit der Ergebnisse auch davon ab, ob die Berech-

nung innerhalb eines vorgegebenen Zeitintervalls abgeschlossen ist. Man unterscheidet zwischen harten und weichen Echtzeitanforderungen. In Systemen mit harten Echtzeitanforderungen kann das Nicht-Einhalten von Zeitschranken zu einem Ausfall des gesamten Systems führen und schwerwiegende Folgen haben. So könnte durch eine Fehlfunktion in einer Fahrzeugsteuerung beispielsweise ein Unfall verursacht werden. Bei weichen Echtzeitanforderungen dagegen verursachen nicht eingehaltene Zeitschranken keine Schäden, und das System läuft weiter. Ein Beispiel für weiche Echtzeitanforderungen sind Multimedia-Anwendungen, bei denen durch nicht rechtzeitig zur Verfügung stehende Audio- oder Videodaten lediglich die Qualität verringert wird.

Um das Einhalten von Zeitschranken garantieren zu können, muss die maximale Programm-Laufzeit (*worst case execution time, WCET*) bekannt sein. Bisher wird die WCET häufig durch Abschätzungen auf Basis von Messungen der Laufzeit bei Worst-Case Eingaben bestimmt. Eine verlässlichere Methode ist das Bestimmen der WCET durch statische Programmanalyse. Dieses Gebiet ist schon seit einigen Jahren Gegenstand der Forschung, und heute stehen bereits verschiedene kommerzielle Werkzeuge zu Verfügung. Ein weiterer Vorteil exakter WCET-Werte ist die Möglichkeit, den Energieverbrauch zu reduzieren, indem die Taktfrequenz des Prozessors soweit gesenkt wird, dass alle Zeitschranken gerade eingehalten werden.

In der Vergangenheit wurde Software für eingebettete Systeme häufig in Assembler entwickelt, um möglichst effizienten Code zu erhalten und so den Anforderungen eingebetteter Systeme, z.B. an Codegröße oder Laufzeit, gerecht zu werden. Durch die ständig wachsende Komplexität der Software werden heute aber hauptsächlich Hochsprachen wie C oder C++ zur Softwareentwicklung eingesetzt. Dadurch sind die Anforderungen an Compiler gestiegen, da diese nun das Erstellen von optimiertem Maschinencode, der möglichst die Effizienz handgeschriebenen Assemblercodes erreichen soll, übernehmen müssen.

1.1 Motivation

Nahezu alle modernen Compiler führen neben dem eigentlichen Übersetzen eines Quellprogramms in Maschinencode eine Reihe verschiedener Programmtransformationen durch, um möglichst effizienten Code für die jeweilige Zielarchitektur zu erzeugen. In den vergangenen Jahrzehnten wurde eine Vielzahl verschiedener Programmtransformationen zur Codeoptimierung entwickelt und in Compiler integriert.

Wann der generierte Code als effizient angesehen wird, ist allerdings nicht eindeutig und abhängig vom jeweiligem Einsatzgebiet. Es sind verschiedene Kriterien denkbar, bezüglich denen der Code optimiert werden kann. Häufig führt eine Optimierung bezüglich eines Kriteriums zu einer Verschlechterung bezüglich eines anderen, so dass entweder ein Optimierungsziel gewählt oder ein Kompromiss zwischen verschiedenen gefunden werden muss. Die größte Bedeutung hat bisher die Opti-

mierung der durchschnittlichen Programm-Laufzeit (*average case execution time, ACET*), sowie auch die Optimierung der Codegröße. Die meisten Compiler unterstützen heute diese Optimierungsziele, und in diesem Bereich wurde auch die meiste Forschung betrieben.

Durch die ständig zunehmende Verbreitung von eingebetteten Systemen in mobilen Geräten und die wachsenden Ansprüche an diese Systeme, wurde auch die Reduzierung des Energieverbrauchs ein wichtiges Thema. Mit dem *encc* Energy Aware C Compiler [SW02] wurde ein Compiler entwickelt, der die Erzeugung von möglichst energie-effizientem Maschinencode zum Ziel hat.

Wie bereits erwähnt, ist bei Echtzeitsystemen die Worst-Case Laufzeit von größerer Bedeutung als die durchschnittliche Laufzeit. Eine Optimierung der durchschnittlichen Laufzeit durch den Compiler würde für solche Systeme kaum Vorteile bringen, falls nicht auch die WCET reduziert wird. Der Einfluss der bisher zur Reduzierung der durchschnittlichen Laufzeit eingesetzten Compiler-Optimierungen auf die WCET ist allerdings bisher größtenteils unbekannt. So wäre es zum Beispiel denkbar, dass Optimierungen, die zu einer Reduzierung der ACET führen, auf die WCET keinen Einfluss haben oder umgekehrt. Es wäre also wünschenswert, als neues Optimierungsziel die Reduzierung der WCET in einen Compiler zu integrieren.

Zu diesem Zweck wird am Lehrstuhl Informatik XII der Universität Dortmund derzeit eine Compiler-Umgebung entwickelt, die die Integration von Compiler und WCET-Analyse zum Ziel hat [FLT06a, FLT06b]. Durch die damit erreichte Kommunikation zwischen Compiler und WCET-Analyse wird es möglich, Compiler-Optimierungen zu integrieren, die WCET-Informationen nutzen können. Desweiteren können auch während der Compilierung, insbesondere in den Optimierungs-Phasen, gewonnene Informationen an die WCET-Analyse weitergegeben werden, um deren Ergebnisse zu verbessern.

1.2 Ziele der Arbeit

In dieser Arbeit sollen in einem ersten Schritt Faktoren ermittelt werden, die einen Einfluss auf die WCET-Analyse haben. Eine Klassifizierung bewährter Compiler-Optimierungen bezüglich ihres Einflusses auf diese Faktoren soll die Auswahl von Optimierungen für eine genauere Untersuchung unterstützen.

Einige nach dieser ersten Klassifizierung ausgewählte Optimierungen sollen anschließend detailliert hinsichtlich ihres Einflusses auf die Worst-Case Laufzeit von Programmen untersucht werden, um so herauszufinden, welche dieser Optimierungen besonders gut, oder aber auch weniger gut, zur Integration in einen WCET-optimierenden Compiler geeignet sind. Dabei soll auch untersucht werden, ob und wie Informationen, die während der WCET-Analyse gewonnen wurden, bei der Durchführung der einzelnen Optimierungen genutzt werden können, und ob während der Optimierung gewonnene Informationen über das Programm zur Verbesse-

rung der WCET-Analyse verwendet werden können.

Die zu untersuchenden Optimierungen sollen bei einigen, für eingebettete Systeme relevanten, Benchmarks für verschiedene Architekturen durchgeführt werden, um anschließend die Auswirkungen auf die ermittelte WCET zu analysieren und mit den Änderungen bei der durchschnittlichen Laufzeit zu vergleichen.

1.3 Verwandte Arbeiten

Während die Optimierung der durchschnittlichen Laufzeit sowie auch der Codegröße und des Energieverbrauchs lange wichtige Forschungsgebiete sind, existieren bisher nur wenige Arbeiten, die sich mit der Optimierung der Worst-Case Laufzeit beschäftigen.

In [ZKW+04, ZWHM04, ZKW+05] werden WCET-Informationen eines Timing-Analysers zur Durchführung von Optimierungen genutzt. Hierzu wurde ein proprietärer Timing-Analyser in eine interaktive Compiler-Umgebung integriert. Dem Compiler ist es damit möglich, durch Aufruf des Timing-Analysers die WCET aller Pfade einer Funktionen oder einer Schleife zu bestimmen. In [ZKW+04] wird ein genetischer Algorithmus eingesetzt, um eine Abfolge von Standard-Optimierungen zu ermitteln, die zu einer minimalen WCET führt. Die Fitnesswerte der möglichen Optimierungssequenzen basieren dabei auf der WCET und der Codegröße des Programms nach der jeweiligen Optimierungssequenz. In der zweiten Arbeit [ZWHM04] werden mit Hilfe von WCET-Informationen die Basisblöcke eines Progamms so angeordnet, dass die Anzahl der Sprünge auf dem Worst-Case Pfad (WC Pfad) minimiert wird. Dieser Ansatz wurde in [ZKW+05] erweitert, indem durch Transformationen wie das Duplizieren von Programmteilen oder Loop Unrolling der WC Pfad weiter optimiert wird.

Die in [LLPM03] vorgestellte Technik entscheidet anhand von WCET-Informationen, ob Basisblöcke durch schnelleren 32-bit Code oder durch platzsparenden 16-bit Thumb-Code eines ARM Prozessors realisiert werden. Somit ist eine Abwägung zwischen Codegröße und Worst-Case Laufzeit möglich.

Ziel der in [Pus02] vorgestellten Technik ist es, die Analysierbarkeit von Programmen zu vereinfachen. Es wird gezeigt, wie Programme so transformiert werden können, dass diese nur noch einen Ausführungspfad besitzen. Damit wird die WCET-Analyse trivial, da keine Pfad-Analyse mehr notwendig ist. Allerdings kann die Ausführungszeit erheblich steigen, weshalb eine Anwendung auf ganze Programme kaum sinnvoll scheint. Eine Transformation bestimmter Abschnitte könnte aber möglicherweise Vorteile bringen.

Auch in [NRM04] soll die WCET Analyse vereinfacht werden. Es werden Kontrollfluss-Transformationen beschrieben, die die Anzahl der Pfade in einem Programm erheblich verringern können und so die Pfad-Analyse vereinfachen und möglicher-

weise den Fehler bei der WCET-Abschätzung verringern. Es wurden allerdings keine Ergebnisse von Experimenten mit einem WCET-Analyse Werkzeug angegeben.

Bei der in [WM04] vorgestellten Optimierung werden bestimmte Funktionen und Datenelemente in einem schnellen Scratchpad-Speicher platziert. Dabei konnten deutliche Verbesserungen der WCET erreicht werden, die sogar höher sind als die prozentualen Verbesserungen der durchschnittlichen Laufzeit. Allerdings wird hier die Auswahl der Objekte, die in dem Scratchpad-Speicher platziert werden, nicht anhand von WCET-Informationen getroffen, sondern auf Basis des Energieverbrauches.

1.4 Struktur der Arbeit

Der restliche Teil dieser Arbeit ist wie folgt strukturiert:

- **Kapitel 2** führt einige für diese Arbeit relevante Grundlagen zu Compileroptimierungen und WCET-Analyse ein. Außerdem werden die verwendeten Werkzeuge sowie die betrachteten Architekturen und Benchmarks vorgestellt.
- **Kapitel 3** stellt eine Klassifizierung verschiedener Compiler-Optimierungen bezüglich ihres möglichen Einflusses auf das Ergebnis der WCET-Analyse vor.
- **Kapitel 4** enthält eine detaillierte Untersuchung dreier verschiedener Kontrollflussoptimierungen bezüglich des Einflusses auf die Worst-Case Laufzeit.
- **Kapitel 5** präsentiert die Ergebnisse der Untersuchung einer Worst-Case Pfad Optimierung.
- **Kapitel 6** untersucht den Einfluss ganzer Optimierungs-Sequenzen, bestehend aus einer Reihe verschiedener Standard Optimierungen eines existierenden Compiler-Frontends, auf die WCET.
- **Kapitel 7** fasst diese Arbeit zusammen und gibt einen Ausblick auf mögliche weitere Arbeiten zu diesem Thema.

2 Grundlagen

In den ersten Abschnitten dieses Kapitels werden einige Grundlagen zu Compiler-Optimierungen sowie zur WCET-Analyse eingeführt. Anschließend wird das in dieser Arbeit verwendete WCET-Analyse Werkzeug aiT vorgestellt und die betrachteten Architekturen, einschließlich der verwendeten Entwicklungswerkzeuge, beschrieben. Der letzte Abschnitt enthält eine Übersicht und kurze Beschreibung aller verwendeten Benchmark-Programme.

2.1 Compiler-Optimierungen

Ein Compiler ist ein Computer-Programm, das ein in einer Quellsprache geschriebenes Programm in ein äquivalentes Programm in einer Zielsprache übersetzt [ASU86]. Üblicherweise wird ein in einer Hochsprache wie C oder C++ geschriebenes Programm in ein Assembler- bzw. Maschinenprogramm für einen bestimmten Prozessor übersetzt.

Wie bereits in der Einleitung angesprochen, werden insbesondere im Bereich der eingebetteten Systeme hohe Ansprüche an einen Compiler gestellt. Der generierte Maschinencode soll möglichst die Effizienz von handgeschriebenem Assemblercode erreichen. Deshalb führen heutige Compiler neben der eigentlichen Übersetzung eine Vielzahl von Optimierungen durch.

2.1.1 Struktur eines Compilers

Die meisten modernen Compiler sind in zwei Phasen aufgeteilt: dem Frontend (Analysephase) und dem Backend (Synthesephase). Als Schnittstelle zwischen den beiden Teilen dient eine, üblicherweise plattform-unabhängige, Zwischendarstellung (*intermediate representation, IR*). Dadurch ist es möglich, ein Frontend für eine bestimmte Sprache mit Backends für unterschiedliche Zielarchitekturen zu kombinieren, wodurch die Portierung auf neue Architekturen erheblich vereinfacht wird. Beide Teile lassen sich wiederum in mehrere Phasen unterteilen. Abbildung 2.1 zeigt den typischen Aufbau eines solchen Compilers.

Das Frontend überführt das Quellprogramm in eine Zwischendarstellung, die möglichst unabhängig von der Quellsprache ist, und prüft dabei auf syntaktische und semantische Korrektheit. Im Einzelnen werden folgende Schritte durchgeführt:

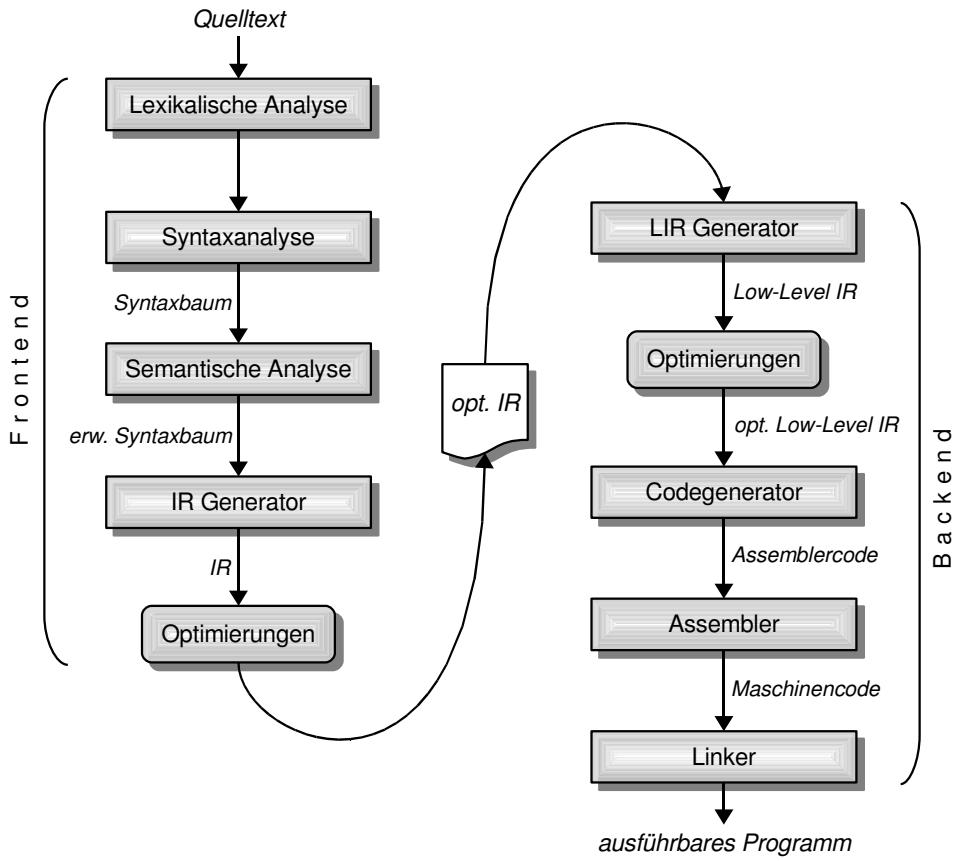

Abbildung 2.1: Aufbau eines Compilers

- **Lexikalische Analyse (Scanner)**
  Die lexikalische Analyse wandelt das Quellprogramm, das als eine Folge von Zeichen eingelesen wird, in eine Folge von Symbolen der Programmiersprache um.

- **Syntaxanalyse (Parser)**
  Bei der Syntaxanalyse wird aus der Folge von Symbolen entsprechend der Grammatik der Sprache ein Syntaxbaum erstellt. In dieser Phase werden syntaktische Fehler wie z.B. fehlende Klammern erkannt.

- **Semantische Analyse**
  Die semantische Analyse überprüft das Programm auf semantische Korrektheit, z.B. Typkorrektheit, und erweitert den Syntaxbaum um semantische Informationen.

- **Zwischendarstellung generieren**
  In der letzten Phase wird aus dem Syntaxbaum eine spezielle Zwischendarstellung des Compilers erzeugt.

Die so erzeugte Zwischendarstellung des Programms wird an das Backend übergeben, welches in folgenden Schritten ausführbaren Maschinencode erzeugt:

- **Low-Level Zwischendarstellung generieren**
  Häufig wird im Backend eine weitere maschinen-abhängige Zwischendarstellung verwendet, die in dieser Phase aus der Zwischendarstellung des Frontend erzeugt wird. Dadurch kann die Durchführung von Optimierungen im Backend erleichtert werden.

- **Codegenerator**
  In dieser Phase wird aus der Zwischendarstellung Assemblercode generiert. Diese Phase lässt sich in die drei Teilprobleme Instruktionsauswahl, Registerallokation und Instruktionsanordnung unterteilen. Bei der Instruktionsauswahl (*code selection*) gilt es, zu jeder Anweisung der Zwischendarstellung eine möglichst günstige Folge von Maschineninstruktionen zu finden. Die Registerallokation hat die Aufgabe, eine möglichst günstige Zuordnung von Variablen zu physikalischen Prozessorregistern zu finden. D.h. es wird festgelegt, welche Variable zu welchem Zeitpunkt in einem Register gehalten oder in den Speicher ausgelagert wird (*spilling*). Bei der Instruktionsanordnung (*instruction scheduling*) werden die Instruktionen schließlich so angeordnet, dass die Ausführungseinheiten des Prozessors möglichst gut ausgelastet werden.

  Bei der Verwendung einer Low-Level Zwischendarstellung im Backend müssen diese Schritte teilweise auch schon bei der Erstellung dieser Zwischendarstellung durchgeführt werden.

- **Assembler**
  Der Assembler übersetzt den generierten Assemblercode in verschiebbaren Maschinencode.

- **Linker**
  Der Linker erzeugt aus einer oder mehreren Objekt-Dateien mit verschiebbarem Maschinencode das ausführbare Programm. Dabei werden auch die endgültigen Adressen für Daten- und Code-Objekte festgelegt.

2.1.2 Optimierungen

Während des Compilier-Vorganges werden an mehreren Stellen verschiedene Programmtransformationen durchgeführt, um die Effizienz des generierten Maschinencodes zu steigern. Diese Transformationen werden üblicherweise als Codeoptimierungen bezeichnet. Die meisten dieser Optimierungen werden auf der vom Frontend des Compilers generierten Zwischendarstellung durchgeführt. Da diese Optimierungs-Phase nicht direkt zum Frontend, aber auch nicht zum Backend gehört, wird gelegentlich auch der Begriff Middleend verwendet. Die an dieser Stelle durchgeführten

Optimierungen sind im Wesentlichen unabhängig von der Zielplattform und werden auch als High-Level Optimierungen bezeichnet. Einige Beispiele sind:

- **Auswerten von Ausdrücken zur Compilezeit**
  Ausdrücke, die nur Konstanten enthalten, können durch den Compiler berechnet und durch eine Konstante ersetzt werden.
- **Eliminierung von totem Code**
  Codeabschnitte, die niemals ausgeführt werden, können aus dem Programm entfernt werden.
- **Inlining**
  Für kleine, häufig aufgerufene Funktionen kann es von Vorteil sein, statt eines Funktionsaufrufes direkt den Programmcode der Funktion einzufügen, um so den zusätzlichen Aufwand, der bei einem Funktionsaufruf entsteht, zu vermeiden.
- **Schleifentransformationen**
  Es existiert ein Vielzahl verschiedener Optimierungen, die versuchen durch das Umstrukturieren von Schleifen deren Ausführung zu verbessern. Da meist ein wesentlicher Teil der Programmlaufzeit in Schleifen verbracht wird, haben diese Optimierungen oft einen großen Einfluss.

Eine weitere Optimierungsphase findet im Backend des Compilers auf der Low-Level Zwischendarstellung oder dem Assemblercode statt. Diese Low-Level Optimierungen können nun auch Informationen über die Zielplattform verwenden, um speziell für diese optimierten Code zu generieren. Einige der High-Level Optimierungen, wie das Auswerten von Ausdrücken zur Compilezeit oder das Eliminieren von totem Code, können auf Assemblerebene ein weiteres Mal durchgeführt werden. Andere Low-Level Optimierungen sind z.B.:

- **Strength Reduction**
  Bestimmte Operationen können durch andere, äquivalente Operationen ersetzt werden, die auf der Zielhardware schneller ausgeführt werden. So könnte z.B. eine Multiplikation durch eine Schiebeoperation ersetzt werden.
- **Software Pipelining**
  Beim Software Pipelining werden die Instruktionen mehrerer Schleifeniterationen überlappend ausgeführt, um die Auslastung der Ausführungseinheiten zu verbessern.

2.2 WCET-Analyse

In Echtzeitsystemen müssen bei der Programmausführung zeitliche Bedingungen eingehalten werden, die oft durch die Umgebung vorgegeben sind. Um dies gewähr-

leisten zu können, muss eine obere Schranke für die Ausführungszeit der Programme bekannt sein. Diese meist als Worst-Case Laufzeit bezeichneten Laufzeit-Schranken sind Voraussetzung für die Durchführung des Scheduling bei Systemen mit vorgegebenen Zeitschranken. Um eine Überdimensionierung der Hardware und die damit verbundenen Kosten zu vermeiden, sollte die Schätzung der oberen Schranke möglichst gut sein, d.h. die Überschätzung gering. Gleichzeitig muss die Schätzung aber sicher sein, d.h. es darf niemals zu einer Unterschätzung kommen.

Bisher wird die WCET oft durch sog. dynamische Analyse ermittelt. Dabei wird die Ausführungszeit eines Programms entweder mit Hilfe eines Prozessorsimulators oder durch Messungen an realer Hardware bestimmt. Dabei wird allerdings nur jeweils ein möglicher Ausführungspfad untersucht. Da eine Messung für alle möglichen Eingabewerte i.d.R. unmöglich ist, muss eine sog. Worst-Case Eingabe ermittelt werden, die zur maximalen Laufzeit führt. Bei komplexeren Programmen ist es aber oft nahezu unmöglich, die Auswirkungen der Eingabedaten auf die Laufzeit zu überschauen, zumal die Hardware moderner Prozessoren, wie z.B. Caches, weitere nur schwer vorhersehbare Auswirkungen auf die Laufzeit haben kann. Um das Risiko der Unterschätzung der WCET bei der dynamischen Analyse zu verringern, werden die gemessenen Werte oft noch mit einem Sicherheitsfaktor multipliziert, womit die Sicherheit der Abschätzung aber immer noch nicht garantiert werden kann und zudem die Überschätzung in den meisten Fällen erhöht wird.

Eine zuverlässigere Methode stellt die WCET-Abschätzung mittels statischer Programm-Analyse dar. Dabei wird das Programm nicht ausgeführt, sondern seine Struktur statisch analysiert, um so eine obere Schranke für die Ausführungszeit des längsten möglichen Pfades für beliebige Eingaben zu ermitteln. Dazu müssen auch die Ausführungszeiten der einzelnen Basisblöcke bestimmt werden. Bei älteren Mikrocontrollern wie dem Intel 8051 [Int94] sind die Ausführungszeiten der Instruktionen konstant, so dass lediglich die Ausführungszeiten aller Instruktionen eines Basisblocks addiert werden müssen. Bei modernen Prozessoren ist die Berechnung allerdings komplizierter, da Eigenschaften wie Pipelining, Sprungvorhersage oder Caches zu variablen Ausführungszeiten der Instruktionen führen, die nur im jeweiligen Kontext bestimmt werden können. Da der Performancegewinn durch diese Prozessorkomponenten heute aber selbst für Echtzeitsysteme unverzichtbar ist, wurden in den vergangen Jahren Methoden entwickelt, um auch für solche Systeme gute WCET-Abschätzungen zu berechnen [OS97, SF99, FMWA99]. Heute existieren einige WCET-Analyse Werkzeuge für verschiedene aktuelle Architekturen.

Die meisten dieser Werkzeuge sind für die Analyse von in C geschriebenen Programmen entwickelt worden. Die WCET-Analyse selbst wird aber auf Assembler- bzw. Maschinencodeebene durchgeführt, da sich nur so exakte Ausführungszeiten bestimmen lassen. Bei der WCET-Analyse werden einige Informationen über den Kontrollfluss des Programms benötigt, wie z.B. die maximale Anzahl an Schleifeniterationen, die maximale Rekursionstiefe oder Adressen von Sprungzielen, die zur Laufzeit berechnet werden. Durch zusätzliche Informationen z.B. über Pfade, die nie

ausgeführt werden oder die relative Ausführungsanzahl bestimmter Pfade, kann die WCET-Abschätzung u.U. verbessert werden. Es gibt verschiedene Ansätze, an diese Informationen zu gelangen. Durch eine Analyse des Objektcodes können einige dieser Informationen gewonnen werden. Weitere müssen aber oft manuell angegeben werden, was einige Kenntnisse des zu analysierenden Programms voraussetzt und sehr fehleranfällig ist. Da der Compiler bereits ausgiebige Analysen auf dem Programm durchführt und dadurch über für die WCET-Analyse relevante Informationen verfügt, die zum Teil nicht mehr aus dem endgültigen Objektcode rekonstruiert werden können, wäre es wünschenswert, wenn diese Informationen an die WCET-Analyse weitergegeben werden könnten. Mit [BH03] wurde eine Liste von Informationen zusammengestellt, die ein Compiler wünschenswerterweise zu Verfügung stellen sollte, um die WCET-Analyse zu erleichtern. Ein weiterer Ansatz besteht darin, die WCET-Analyse in einen Compiler zu integrieren, um so einen direkten Informationsaustausch zu ermöglichen [FLT06a, FLT06b].

Trotz einiger Unterschiede bei der Implementierung in den verschiedenen Werkzeugen lässt sich die WCET-Analyse grob in drei Teilaufgaben unterteilen: das Zusammenstellen von Kontrollflussinformationen, die Berechnung von Ausführungszeiten der Basisblöcke sowie das Bestimmen eines WC Pfades.

2.2.1 Kontrollflussanalyse

Ausgangspunkt für die statische WCET-Analyse ist ein Kontrollflussgraph des zu analysierenden Programms. Ist die WCET-Analyse in einem Compiler integriert, so kann der Kontrollflussgraph leicht durch den Compiler erstellt werden. Steht dem WCET-Analyse Werkzeug lediglich der Objektcode des Programms zur Verfügung, so muss der Kontrollflussgraph durch Disassemblieren des Maschinencodes rekonstruiert werden. Dies ist jedoch nicht in allen Fällen problemlos möglich. Werden beispielsweise Sprungziele erst zur Laufzeit berechnet, z.B. häufig bei switch-case-Anweisungen, so ist die Kontrollflussstruktur aus dem Assemblercode nicht rekonstruierbar. Die möglichen Sprungadressen müssen in einem solchen Fall durch den Benutzer angegeben werden.

Desweiteren muss für die WCET-Analyse die maximale Anzahl an Iterationen für alle Schleifen sowie die maximale Rekursionstiefe von rekursiven Funktionen bekannt sein. Einige Werkzeuge versuchen die Anzahl der Schleifeniterationen automatisch zu bestimmen, insbesondere bei komplizierteren Schleifen sind allerdings oft manuelle Angaben notwendig.

Um die WCET-Abschätzung zu verbessern und die Komplexität der Analyse zu verringern, wird zu Beginn der Analyse oft versucht, Pfade zu bestimmen, die bei einer Ausführung des Programms nicht durchlaufen werden (*infeasible paths*). Für den folgen Programmausschnitt könnte beispielsweise ermittelt werden, dass der *then*-Teil der *if*-Anweisung in den ersten 10 Iterationen der *for*-Schleife nicht ausgeführt wird. Die entsprechenden Pfade müssten somit bei der Bestimmung des WCET-

Pfades nicht berücksichtigt werden.

```
for (i=0; i<20; i++) {
  if (x >= 10) {
    ...
  }
  ...
}
```

Für diese Analyse ist es notwendig, Sprungbedingungen statisch auszuwerten. Dazu müssen z.B. mittels abstrakter Interpretation [CC77] die Werte von Variablen zu beliebigen Programmpunkten ermittelt werden.

2.2.2 Berechnung der Ausführungszeiten

Nachdem die Kontrollstruktur des Programms und damit die möglichen Ausführungspfade bekannt sind, müssen im zweiten Schritt der WCET-Analyse die maximalen Ausführungszeiten der einzelnen Basisblöcke berechnet werden. Für diesen Analyseschritt wird häufig auch der Begriff Low-Level Analyse verwendet. Es sind detaillierte Kenntnisse der verwendeten Hardware nötig, wie z.B. die Taktfrequenz, Speicherzugriffszeiten, vorhandene Cache-Speicher und deren Organisation, Pipeline-Verhalten, usw.

In [EES+99] wird die Low-Level-Analyse unterteilt in die Analyse globaler sowie lokaler Auswirkungen auf die Laufzeit. Globale Auswirkungen haben beispielsweise Caches, dynamische Sprungvorhersage oder *translation lookaside buffers*. Zu der Analyse der globalen Auswirkungen gehört somit auch die Analyse des Cacheverhaltens, bei der die Speicherzugriffe des Programms als *cache hit* oder *cache miss* klassifiziert werden. Ist eine exakte Bestimmung nicht möglich, muss eine Worst-Case Annahme gemacht werden. Bei der Analyse der lokalen Auswirkungen geht es um Effekte, die nur eine Instruktion und die benachbarten Instruktionen betreffen, wie z.B. Pipeline-Effekte.

Bei einigen komplexen Prozessoren ist diese Trennung allerdings nicht mehr möglich. Durch sog. *timing anomalies* kann es vorkommen, dass eine lokale Worst-Case Entscheidung, wie z.B. ein *cache miss*, global zu einer besseren Laufzeit führt [RWT+06]. Für eine korrekte WCET-Abschätzung müssen hier immer sämtliche Möglichkeiten betrachtet werden.

Sowohl die Cache-Analyse als auch die Pipeline-Analyse kann mittels abstrakter Interpretation durchgeführt werden, wie in [FMWA99] und [SF99] vorgestellt. Eine weitere in [EES+99] vorgeschlagene Möglichkeit zur Bestimmung der Ausführungszeiten ist die Simulation einzelner Basisblöcke mit einem Prozessorsimulator unter Vorgabe eines Ausführungsszenarios, d.h. Angaben wie z.B. ob ein Speicherzugriff zu einem *cache-hit* oder *cache-miss* führt.

2.2.3 Bestimmen des Worst-Case Pfades

Als letzter Schritt der WCET-Analyse wird auf Basis des Kontrollflussgraphens, zusätzlicher Kontrollflussinformationen, die während der Kontrollflussanalyse gewonnen wurden oder durch den Benutzer bzw. den Compiler angegeben wurden, sowie der maximalen Ausführungszeiten der Basisblöcke ein sogenannter Worst-Case Pfad (WC Pfad) ermittelt, der zu einer maximalen Ausführungszeit führt. Dazu sind die drei folgenden Berechnungsverfahren gebräuchlich:

- **Baum-basierte Berechnung**
  Bei der Baum-basierten Berechnung wird ein Syntaxbaum des Programms *bottom-up* durchlaufen. Die Knoten des Baums beschreiben die Kontrollstruktur wie z.B. Schleifen, if-Anweisungen oder Sequenzen, die Blätter repräsentieren die Basisblöcke. Für die verschiedenen Knotentypen werden Regeln aufgestellt, wie die WCET für diesen Knotentyp aus der WCET der Kind-Knoten berechnet werden kann.

  Dieses Verfahren ist einfach und effizient durchführbar, allerdings ist es schwierig, Kontrollflussinformationen zu berücksichtigen, die z.B. Abhängigkeiten zwischen verschiedenen Programmpfaden beschreiben. In [LBJ+95] wird beispielsweise ein Baum-basiertes Berechnungsverfahren verwendet.

- **Pfad-basierte Berechnung**
  Die Pfad-basierte Berechnung bestimmt die WCET für verschiedene mögliche Pfade eines Programms, um so den längsten dieser Pfade zu ermitteln. Dabei wird meist in einem *bottom-up*-Verfahren erst der längste Pfad aufgerufener Funktionen und innerer Schleifen bestimmt und die ermittelte WCET anschließend zur Ermittlung des längsten Pfades der aufrufenden Funktion oder äußeren Schleife genutzt. Enthält ein Programmabschnitt sehr viele aufeinander folgende *if*-Anweisungen, so kann die Anzahl der Pfade schnell sehr groß werden und die Berechnung entsprechend aufwendig.

  Ein Pfad-basiertes Berechnungsverfahren wird beispielsweise in den Arbeiten [ZKW+04, ZWHM04, ZKW+05] verwendet.

- **Implicit Path Enumeration Technique (IPET)**
  Während bei der Pfad-basierten Berechnung jeder Pfad explizit betrachtet wird, werden bei der *implicit path enumeration technique* [LM95] die einzelnen Pfade nur implizit berücksichtigt. Die WC Pfad Berechnung wird dazu als Maximierungsproblem formuliert, wobei die Nebenbedingungen die Kontrollstruktur des Programms beschreiben, und die Zielfunktion die WCET des gesamten Programms berechnet. Am häufigsten wird dazu ganzzahlige lineare Optimierung verwendet, da hierfür Programme existieren, die meist in kurzer Zeit eine Lösung berechnen. In [OS97] wurde eine Formulierung als *constraint satisfaction problem (CSP)* vorgestellt. Dadurch sind komplexere Nebenbedingungen möglich, das Berechnen einer Lösung ist aber u.U. zeitaufwendiger.

Ein Vorteil dieses Berechnungsverfahrens ist, dass leicht zusätzliche Einschränkungen der möglichen Kontrollflusspfade berücksichtigt werden können, indem diese als Nebenbedingungen des Maximierungsproblems formuliert werden.

2.3 aiT – AbsInt's WCET-Analyse Werkzeug

In den letzten Jahren wurden zwar einige verschiedene akademische WCET-Analyse Werkzeuge entwickelt, bis heute existieren jedoch nur sehr wenige kommerzielle Anwendungen. Die existierenden Werkzeuge unterscheiden sich in den unterstützten Prozessoren und Compilern sowie in den verwendeten Berechnungsmethoden.

Ein sehr fortgeschrittenes Programm ist das WCET-Analyse Werkzeug aiT der Firma AbsInt Angewandte Informatik GmbH, Saarbrücken [Abs06a]. Es berechnet eine WCET-Abschätzung durch statische Programm-Analyse und berücksichtigt dabei auch Eigenschaften moderner Prozessoren wie Pipelining, Sprungvorhersage und Caches. Dabei werden verschiedene Analyseschritte mittels abstrakter Interpretation durchgeführt und der WCET-Pfad durch ein IPET-basiertes Verfahren bestimmt. aiT ist für eine Reihe verschiedener Prozessoren verfügbar. Zur Zeit sind dies: ARM7, HCS12/STAR12, PowerPC 555/565/755, C166/ST10, TMS320C3x und TriCore 1796. Für diese Arbeit wurden die aiT Versionen für die ARM7 Prozessorfamilie und den Infineon TriCore 1796 Mikrocontroller verwendet.

2.3.1 Ablauf der Analyse

Als Eingabe für die WCET-Analyse mit aiT dient ein compiliertes und gelinktes ausführbares Programm sowie eine Konfigurationsdatei mit Benutzerangaben zur verwendeten Hardware und dem zu analysierenden Programm (siehe Abschnitt 2.3.2). aiT wurde für die Analyse von in C geschriebenen Programmen entwickelt. Allerdings müssen einige Einschränkungen beachtet werden: das zu analysierende Programm darf keine dynamischen Datenstrukturen und keine `setjmp`/`longjmp` Anweisungen enthalten. Die ANSI-C Anweisungen `setjmp` und `longjmp` werden üblicherweise zur Fehlerbehandlung eingesetzt. Ein Aufruf von `setjmp` speichert den aktuellen Programmstatus, und durch einen späteren Aufruf von `longjmp` kann die Programmausführung an der gespeicherten Stelle fortgesetzt werden. Damit sind Sprünge über Funktionsgrenzen hinweg möglich, die bei der Analyse schwer zu berücksichtigen wären.

Die Analyse wird in mehreren Phasen durchgeführt, der genaue Ablauf ist in Abbildung 2.2 dargestellt. Nach erfolgreicher Analyse werden die Ergebnisse graphisch angezeigt. Neben der WCET des gesamten Programms werden auch die WCET und die Ausführungsanzahl jedes Basisblockes für das ermittelte Worst-Case Szenario angezeigt. Zusätzlich kann für jeden Basisblock eine detaillierte Darstellung des Prozessorzustandes bei jedem Taktzyklus angezeigt werden.

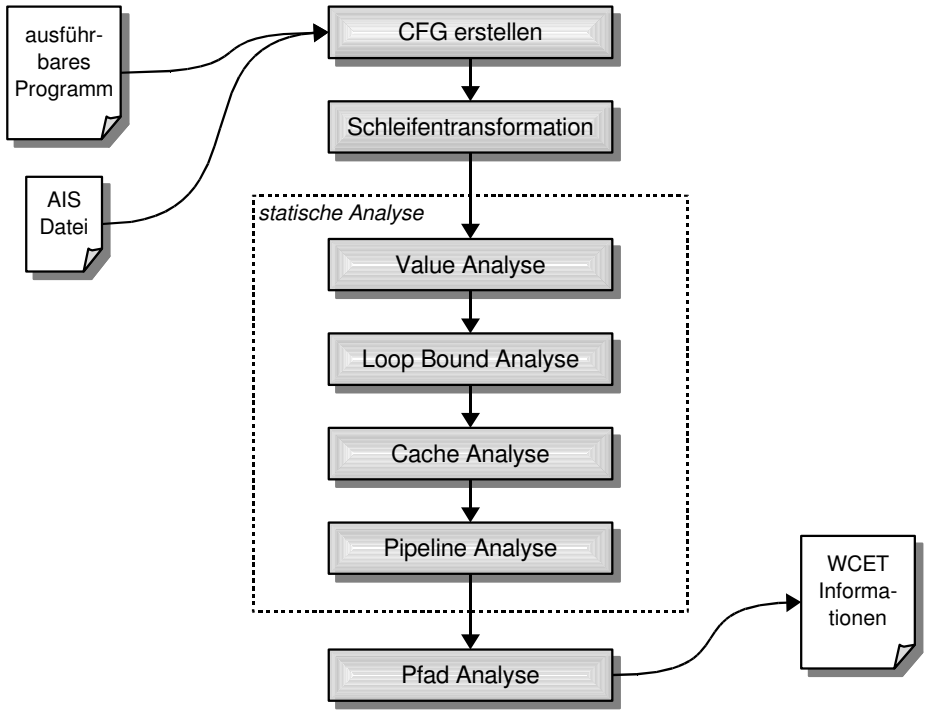

Abbildung 2.2: Ablauf der WCET-Analyse mit aiT

Erstellen des Kontrollflussgraphens

In der ersten Analysephase wird das gegebene Programm disassembliert und der Kontrollflussgraph des Programms rekonstruiert. Dieser Teil der Analyse ist jeweils angepasst an einen bestimmten Compiler. Denn nur mit Kenntnissen über die Art der Codegenerierung des verwendeten Compilers können beispielsweise Sprungziele ermittelt werden, deren Adressen erst zur Laufzeit aus Registerwerten berechnet werden. Trotzdem kommt es vor, dass Sprungziele nicht automatisch bestimmt werden können, diese müssen dann durch den Benutzer angegeben werden.

Der erstellte Kontrollflussgraph wird in AbsInt's CRL-Format (*control flow representation language*) überführt, das als Austauschformat zwischen den Analysephasen dient. In den folgenden Phasen werden der CRL-Darstellung zusätzliche Informationen hinzugefügt.

Schleifentransformation

Die Schleifentransformationsphase wandelt alle im Programm vorkommenden Schleifen in eine äquivalente Darstellung als rekursive Funktion um. Abbildung 2.3 zeigt ein Beispiel für eine solche Transformation in Pseudocode. Die Darstellung als rekursive Funktion kann zu genaueren Ergebnissen führen, da aiT bei der Analyse

verschiedene Kontexte unterscheidet, in denen eine Funktion aufgerufen wird. Nach dieser Transformation stellt jede Iteration einer Schleife einen Funktionsaufruf dar, dem ein eigener Kontext zugeordnet werden kann.

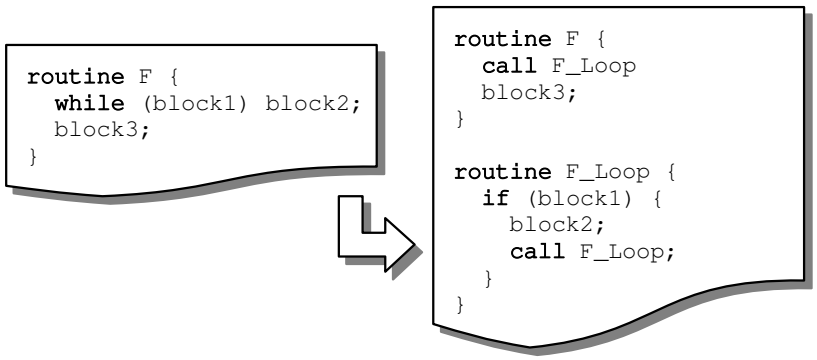

Abbildung 2.3: Schleifentransformation in aiT

Value Analyse

Ziel der Value Analyse ist es, den Wert bzw. einen möglichen Wertebereich der Prozessorregister zu jeden Programmpunkt, in jedem Kontext zu bestimmen. Hierzu wird die Methode der abstrakten Interpretation verwendet. Das Bestimmen eines Registerwertes ist natürlich nur dann möglich, wenn der Wert nur von Konstanten abhängt und nicht von Eingabedaten. aiT kann auch den Wert von Variablen bestimmen, die zwischenzeitlich in den Hauptspeicher ausgelagert werden. Allerdings führt ein Schreibzugriff auf eine unbekannte Speicheradresse dazu, dass sämtliche Informationen über im Hauptspeicher vorhandene Daten verworfen werden müssen, da nicht bestimmt werden kann, welche Speicherstelle tatsächlich verändert wurde.

Die so bestimmten Registerwerte werden in den folgenden Analysephasen für verschiedene Zwecke genutzt. Mit den Werten können Sprungbedingungen ausgewertet werden, um so Pfade zu ermitteln, die in einem bestimmten Kontext nicht durchlaufen werden (*infeasible paths*). Ermittelte Adressen von Speicherzugriffen werden sowohl für die Cache Analyse als auch für die Pipeline Analyse benötigt. Außerdem werden die Registerwerte bei der Loop Bound Analyse verwendet.

Loop Bound Analyse

Bei der Loop Bound Analyse wird versucht, die maximale Anzahl an Iterationen der im Programm vorkommenden Schleifen zu bestimmen. Dabei werden die Ergebnisse der Value Analyse verwendet und mit einer Mustererkennung kombiniert, die versucht Übereinstimmungen mit bestimmten Schleifenmustern zu finden, die der verwendete Compiler üblicherweise generiert.

Die Loop Bound Analyse ist allerdings meist nur für einfache Schleifen erfolgreich. Für kompliziertere Schleifen sind Benutzerangaben notwendig.

Cache Analyse

Die Cache Analyse untersucht den Zustand eines Cache-Speichers während der Programmausführung, um so Zugriffe auf den Hauptspeicher als sicheren *cache-hit* oder unklassifizierten Zugriff zu klassifizieren. Die Cache Analyse verwendet die bei der Value Analyse bestimmten Adressen von Speicherzugriffen und basiert wie die Value Analyse auch auf abstrakter Interpretation.

Pipeline Analyse

Die Pipeline Analyse modelliert das Pipeline-Verhalten des Prozessors, um so die WCET für alle Basisböcke (bzw. die ausgehenden Kontrollflusskanten) in allen möglichen Kontexten zu bestimmen. Wie schon die Value und Cache Analyse basiert auch die Pipeline Analyse auf abstrakter Interpretation. Ausgehend von möglichen Startzuständen, die durch den Vorgänger-Basisblock gegeben sind, wird die Ausführung der Instruktionen des Basisblocks simuliert. Dabei müssen mögliche *pipeline stalls*, unterschiedliche Speicherzugriffszeiten und die Ergebnisse der Cache Analyse berücksichtigt werden. Durch fehlende Informationen während der abstrakten Ausführung können Zustände aufgeteilt werden in mehrere mögliche Folgezustände, die dann bei den weiteren Instruktionen berücksichtigt werden müssen und am Ende des Basisblocks die Menge der Startzustände für nachfolgende Basisblöcke bilden. Diese Simulation wird für alle Basisblöcke wiederholt, bis sich die Zustandsmenge nicht mehr ändert. Schließlich wird für jeden Basisblock die maximal erreichte Anzahl an Taktzyklen bestimmt.

Pfad Analyse

Als letzter Schritt der WCET-Analyse wird mit Hilfe der Ergebnisse der vorherigen Analyseschritte ein Worst-Case Pfad ermittelt. Die Pfad Analyse in aiT basiert auf der *implicit path enumeration technique* (siehe Abschnitt 2.2.3) und ganzzahliger Programmierung (*integer linear programming, ILP*).

Die Pipeline Analyse liefert für jede Kontrollflusskante e und jeden möglichen Ausführungskontext c eine WCET-Abschätzung $T(e, c)$. Wenn die Ausführungsanzahl $C(e, c)$ für jede Kante e in jedem Kontext c bekannt ist, lässt sich die gesamte Ausführungszeit als $\sum_{c,e} C(e, c) \cdot T(e, c)$ berechnen. Ziel der WCET-Analyse ist es eine Belegung der Variablen $C(e, c)$ zu finden, so dass alle Kontrollfluss-Beschränkungen eingehalten werden und die Ausführungszeit maximiert wird. Dazu wird ein ILP-Modell erstellt, dessen Nebenbedingungen über die Variablen $C(e, c)$ sich ergeben

aus der Struktur des Programms, den Loop Bounds, bei der Value Analyse ermittelten *infeasible paths*, sowie Benutzerangaben, die den Kontrollfluss einschränken. Dabei soll die Zielfunktion $\sum_{c,e} C(e,c) \cdot T(e,c)$ maximiert werden. Für dieses Modell wird anschließend mit dem externen Programm `lp_solve` [lps06] eine Lösung berechnet.

Als Ergebnis erhält man die WCET-Abschätzung für das gesamte Programm sowie die Werte $C(e,c)$ aller Kanten und Kontexte, die zu dieser Laufzeit führen. Damit kann auch der WC Pfad bestimmt werden.

2.3.2 Benutzerangaben

Neben dem eigentlichen Programm benötigt aiT als Eingabe eine Konfigurationsdatei, genannt AIS Datei, die Benutzerangaben zur verwendeten Hardware, dem Compiler, einigen Analyseparametern und dem zu analysierenden Programm enthalten kann. Einige der Angaben sind für die Analyse zwingend erforderlich, wie beispielsweise Loop Bounds, die nicht automatisch erkannt werden. Andere sind optional und können zusätzliche Informationen über das Programm liefern, um die WCET-Überschätzung zu verringern. Alternativ ist es auch möglich, einige dieser Angaben als Kommentare im C-Quelltext anzugeben. Im Folgenden werden einige wichtige Benutzerangaben vorgestellt. Ein vollständige Liste findet sich in [Abs06b] bzw. [Abs06c].

Taktfrequenz des Prozessors

Wenn die Taktfrequenz des verwendeten Prozessors angegeben wird, zeigt aiT die WCET des gesamten Programms und der einzelnen Funktionen außer in Taktzyklen auch in Sekunden an.

Speicherlayout und Zugriffszeiten

Verschiedene Speicherbereiche können sehr unterschiedliche Zugriffszeiten benötigen. So könnte z.B. über einen bestimmten Adressbereich ein langsamer Flash-Speicher angesprochen werden und über einen weiteren Bereich schneller DRAM-Speicher. Damit bei der Pipeline Analyse jeweils die korrekte Zugriffszeit berücksichtigt werden kann, muss die genaue Aufteilung des Adressbereiches mit den jeweiligen Zugriffszeiten der einzelnen Bereiche angegeben werden.

Verwendeter Compiler

Die Angabe des verwendeten Compilers ist immer erforderlich. Die ersten Analysephasen von aiT, in denen der Objektcode analysiert wird, sind speziell an bestimmte

Compiler angepasst. Nur mit Hilfe von Informationen über die Art der Codegenerierung des Compilers kann der Kontrollflussgraph des Programms automatisch rekonstruiert werden.

Kontexteinstellungen

Bei der statischen Analyse eines Programms werden abstrakte Informationen für jeden Programmpunkt berechnet. Diese abstrakten Informationen müssen jeden möglichen konkreten Programmzustand an einem Programmpunkt approximieren, egal über welchen Kontrollflusspfad der Programmpunkt erreicht wurde. Die Value Analyse ermittelt beispielsweise mögliche Registerwerte zu jedem Programmpunkt. Wird eine Funktion an verschiedenen Stellen aufgerufen, so müssen bei der Berechnung des möglichen Wertebereiches eines Registers alle möglichen Aufrufe der Funktion berücksichtigt werden.

Um die dadurch entstehenden Ungenauigkeiten zu vermeiden, unterscheidet aiT die verschiedenen Kontexte, in denen eine Funktion aufgerufen wird. Jeder dieser möglichen Kontexte wird bei der Value Analyse, Pipeline Analyse, etc. separat betrachtet. Durch die Transformation von Schleifen in rekursive Funktionen werden auch die Iterationen einer Schleife jeweils in einem eigenen Kontext betrachtet.

Ohne eine Begrenzung würde die Anzahl der Kontexte allerdings schnell enorm groß werden. Aus diesem Grund können bei aiT verschiedene Parameter angegeben werden, um die Anzahl der Kontexte zu beschränken. Jeder Kontext einer Funktion wird durch eine Sequenz von Funktionsaufrufen beschrieben. Der letzte Aufruf in der Sequenz ist immer der Aufruf der betrachteten Funktion. Die Länge dieser *call strings* kann begrenzt werden, so dass nur die letzten Funktionsaufrufe eines *call strings* unterschieden werden. Durch einen zweiten Parameter kann die Anzahl der Schleifeniterationen begrenzt werden, die in einem eigenen Kontext betrachtet werden.

Je mehr Kontexte unterschieden werden, desto genauer wird die WCET-Abschätzung. Allerdings steigt auch die benötigte Rechenzeit mit der Anzahl der Kontexte.

Loop Bounds

Für alle Schleifen, deren Loop Bounds nicht von aiT automatisch ermittelt werden können, müssen diese durch den Benutzer angegeben werden.

Kontrollflussbeschränkungen

aiT bietet die Möglichkeit, durch Annotationen den möglichen Kontrollfluss einzuschränken. Dazu können lineare (Un-)gleichungen angegeben werden, die die Ausführungsanzahl verschiedener Instruktionen zueinander in Beziehung setzen. Diese

Angaben werden bei der Pfad Analyse berücksichtigt, indem daraus zusätzliche Nebenbedingungen für das ILP-Modell generiert werden.

Bei unstrukturierten Schleifen, die von aiT nicht als Schleifen erkannt werden, ist die Angabe solcher Annotationen zur Begrenzung der maximalen Anzahl an Iterationen zwingend erforderlich.

Bekannte Registerwerte

In der Value Analyse wird versucht, Registerwerte zu beliebigen Programmpunkten zu bestimmen. Wenn die Value Analyse für bestimmte Werte nicht erfolgreich war, können diese auch manuell angegeben werden. Es sind dabei allerdings nur Angaben möglich, die für alle Ausführungskontexte einer Instruktion gültig sind.

2.4 Betrachtete Architekturen

Die Untersuchungen dieser Arbeit basieren auf zwei verschiedenen Prozessorarchitekturen, die im Bereich der eingebetteten Systeme weit verbreitet sind: die ARMv4T Architektur, speziell die ARM7 Prozessorfamilien, sowie die TriCore Architektur v1.3 von Infineon. Die betrachteten Prozessoren werden in den beiden folgenden Abschnitten vorgestellt.

2.4.1 ARM7

Die ARM7 Prozessorfamilie beinhaltet verschiedene 32-bit RISC Prozessorkerne, die von ARM Limited als *intellectual property (IP)* vertrieben werden. Durch den niedrigen Energieverbrauch und einen geringen Bedarf an Chipfläche eignen sich die Prozessorkerne der ARM7 Familie besonders für den Einsatz in eingebetteten Systemen. Sie wurden in einer Vielzahl von Mikrocontrollern und *Systems-on-a-chip (SoC)* integriert und machen heute einen großen Anteil der eingebetteten Prozessoren aus. In dieser Arbeit wird der ARM7TDMI [ARM01], der einfachste Prozessorkern dieser Familie, betrachtet.

ARM7TDMI

Der ARM7TDMI hat eine Von-Neumann Architektur mit einem 32-bit Prozessorbus für Daten und Instruktionen. Er unterstützt 8-bit, 16-bit und 32-bit Datentypen. Der Prozessor besitzt 31 universelle 32-bit Register und 6 Statusregister. Davon sind immer nur 16 universelle Register und ein bzw. zwei Statusregister für den Programmierer sichtbar, die je nach Betriebsmodus umgeschaltet werden.

Es steht ein 4 GB Adressraum zur Verfügung. Für diese Arbeit wird sowohl bei der

WCET-Analyse als auch bei der Simulation von einem einzigen homogenen Speicherbereich ausgegangen, der Zugriffe innerhalb eines Taktzyklus erlaubt.

Der ARM7TDMI besitzt eine Instruktions-Pipeline zur Steigerung der Verarbeitungsgeschwindigkeit. Diese ist sehr einfach aufgebaut und besteht lediglich aus drei Stufen (siehe Abbildung 2.4). In der *Fetch*-Phase wird ein Instruktionswort aus dem Speicher geladen. In der *Decode*-Phase wird das Instruktionswort dekodiert und die verwendeten Register bestimmt. In der *Execute*-Phase werden die Register gelesen, ALU Operationen ausgeführt und Ergebnisse in Register geschrieben. Auch Speicherzugriffe werden in der *Execute*-Phase ausgeführt.

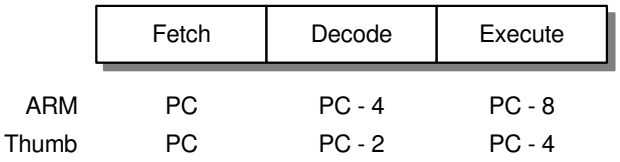

Abbildung 2.4: Die Pipeline des ARM7TDMI

Eine Besonderheit des ARM7TDMI und vieler anderer ARM-Prozessoren sind die zwei unterstützten Instruktionssätze, zwischen denen jederzeit durch spezielle Instruktionen umgeschaltet werden kann: dem 32-bit ARM-Instruktionssatz und dem 16-bit Thumb-Instruktionssatz.

Im ARM-Instruktionssatz sind alle für ARM-Architekturen üblichen Instruktionen enthalten. Jede dieser Instruktionen kann bedingt, d.h. in Abhängigkeit von bestimmten Status-Flags, ausgeführt werden. Dazu stehen 15 verschiedene Bedingungscodes zur Verfügung, die in den ersten vier Bits des Instruktionswortes codiert werden.

Der Thumb-Instruktionssatz wurde eingeführt, um die Codegröße von Programmen zu verringern. Er enthält lediglich eine Auswahl der am häufigsten genutzten Befehle des ARM-Instruktionssatzes. Da lediglich 16 Bit für die Instruktionsworte zur Verfügung stehen, muss beim Thumb-Instruktionssatz auf die bedingte Ausführung beliebiger Befehle verzichtet werden. Auch die Anzahl der verfügbaren Register ist eingeschränkt: es kann nur auf 8 universelle Register direkt zugegriffen werden. Die Thumb-Instruktionen werden vor der Ausführung in ARM-Instruktionen umgewandelt, sodass intern nur 32-bit ARM-Instruktionen verarbeitet werden.

ARM Software Development Toolkit

Das *ARM Software Development Toolkit* [ARM98] ist eine von ARM Ltd. angebotene Entwicklungsumgebung für ARM-Prozessoren. Neben C-Compiler, Assembler, Linker und Debugger ist auch ein Prozessor-Simulator für verschiedene ARM-Prozessoren enthalten.

Die C-Compiler `armcc` für den ARM-Modus und `tcc` für den Thumb-Modus

wurden in dieser Arbeit für die Erstellung aller Programme für den ARM7TDMI verwendet. Der in den Debugger `armsd` integrierte Prozessor-Simulator *ARMulator* wurde verwendet, um die durchschnittliche Laufzeit der untersuchten Benchmark-Programme auf einem ARM7TDMI-Prozessor zu ermitteln.

2.4.2 Infineon TriCore v1.3

Die TriCore Architektur von Infineon wurde speziell für eingebettete Echtzeitsysteme entwickelt. Sie soll die Realzeit-Fähigkeiten eines Mikrocontrollers, die Rechenleistung eines digitalen Signalprozessors (DSP) und die Effizienz einer RISC Load/Store Architektur in einem Prozessorkern vereinen. Es werden sowohl IP Prozessorkerne als auch komplette Mikrocontroller für verschiedene Anwendungsbereiche angeboten. In dieser Arbeit wird der Mikrocontroller TC1796, der für Anwendungen im Automobilbereich und Industriesteuerungen konzipiert ist, betrachtet [Inf05a, Inf05b].

Infineon TC1796

Der Infineon TC1796 enthält neben dem TriCore v1.3 Prozessorkern noch einige weitere Komponenten wie z.B. Timer, A/D-Wandler, einen DMA-Controller und verschiedene Speicher. Da der TC1796 eine Harvard-Architektur besitzt, verfügt er über getrennte Daten- und Instruktionsbusse sowie jeweils eigene Speicher. In Abbildung 2.5 ist die Speicherhierarchie des TC1796 dargestellt.

Der Registersatz des TC1796 ist unterteilt in 16 allgemein verwendbare 32-bit Datenregister, 16 allgemein verwendbare Adressregister, sowie einige Statusregister. Diese Trennung ermöglicht die parallele Ausführung mehrerer Instruktionen innerhalb eines Taktzyklus. Dazu besitzt der TC1796 insgesamt drei Pipelines (siehe Abbildung 2.6). Die Integer- und die Load/Store-Pipeline haben jeweils 4-Stufen. Nach der gemeinsamen *Fetch*-Stufe werden die Instruktionen einer der Pipelines zugeordnet. Die Integer-Pipeline bearbeitet arithmetische Instruktionen und bedingte Sprünge, die Load/Store-Pipeline Speicherzugriffe, Adressarithmetik, unbedingte Sprünge und Funktionsaufrufe. Folgt auf eine Instruktion für die Integer-Pipeline eine Instruktion für die Load/Store-Pipeline, so werden diese parallel ausgeführt. Die dritte Pipeline ist nur für die Ausführung der `loop` Instruktion zuständig. Dadurch sind sog. *zero-overhead loops* [LBSL96] möglich, die keine zusätzlichen Taktzyklen für das Verwalten der Indexvariablen und Prüfen einer Abbruchbedingung benötigen.

Ein weiteres Merkmal des TC1796 ist die integrierte statische Sprungvorhersage. Abhängig von der Art einer Sprunginstruktion wird eine Vorhersage getroffen, ob der Sprung durchgeführt wird oder nicht. Bei einer korrekten Vorhersage kann die Länge der *pipeline stalls*, die durch Sprünge verursacht werden, verringert werden.

Der TriCore Befehlssatz enthält neben den normalen 32-bit Instruktionen auch ei-

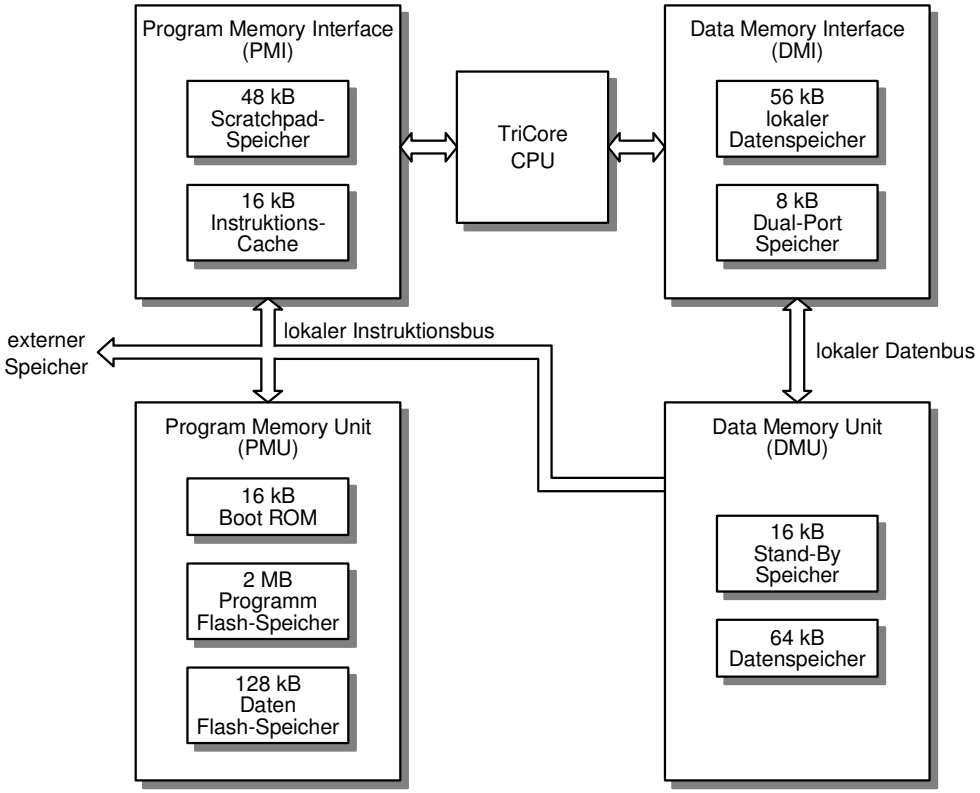

Abbildung 2.5: Speicherhierarchie des Infineon TC1796

nige 16-bit Instruktionen, die an beliebiger Stelle kombiniert mit 32-bit Instruktionen eingesetzt werden können, um die Codegröße und die Energieeffizienz zu verbessern. Alle Instruktionen besitzen eine 32-bit Version. Für die am häufigsten verwendeten Instruktionen gibt es zusätzlich eine 16-bit Variante.

TriCore GNU Toolchain

Die für diese Arbeit verwendeten Entwicklungswerkzeuge für die TriCore Architektur stammen aus einer GNU Toolchain, die von der Firma HighTec EDV-Systeme GmbH für die TriCore Architektur portiert wurde [Hig06]. Sie enthält die üblichen GNU Werkzeuge für die Programmentwicklung in C/C++: den Compiler `tricore-gcc`, Assembler, Linker, Debugger, etc.

Zusätzlich ist ein Prozessorsimulator enthalten, der in Verbindung mit dem Debugger `tricore-gdb` genutzt werden kann. Allerdings werden in der verwendeten Version nur die TriCore Implementierungen Rider-B und Rider-D unterstützt, nicht aber der TC1796. Für die Laufzeitmessungen in dieser Arbeit wurde der Simulator für den Rider-B verwendet. Durch Unterschiede bei den Speicherhierarchien und

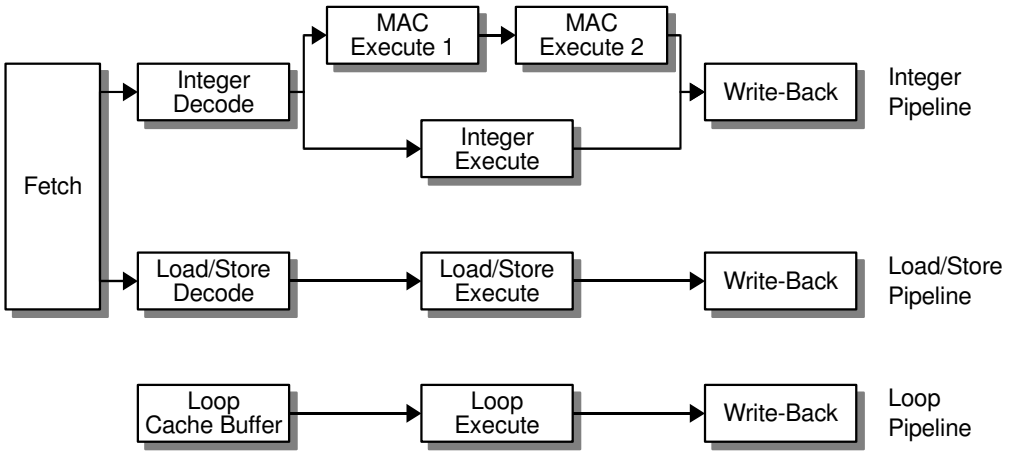

Abbildung 2.6: Die Pipelines des Infineon TC1796

Zugriffszeiten des Rider-B und TC1796 sind die so ermittelten absoluten Laufzeiten nicht direkt vergleichbar mit den Ergebnissen der WCET-Analyse für den TC1796.

2.5 Benchmarks

Zur Bewertung der untersuchten Optimierungen wurden verschiedene Benchmark Programme verwendet, die eine Relevanz für eingebettete Systeme und Echtzeitsysteme besitzen. Viele dieser Benchmarks stammen aus den speziell für eingebettete Systeme entwickelten Benchmark Suiten MediaBench [LPMS97] und MiBench [GRE+01]. Während MediaBench insgesamt 11 Anwendungen aus den Bereichen Multimedia und Kommunikation enthält, repräsentieren die 35 Benchmarks der MiBench Suite Anwendungen aus sechs verschiedenen Anwendungsgebieten.

Diese Benchmarks wurden jedoch nicht speziell für die statische WCET-Analyse entwickelt, so dass meist noch einige Anpassungen nötig waren. Beispielweise lesen die meisten dieser Benchmarks ihre Eingabedaten aus einer Datei und schreiben auch die Ergebnisse wieder in eine Datei. Zusätzlich werden während der Ausführung oft Statusmeldungen ausgegeben. Alle diese Ein-/Ausgabe-Operationen wurden aus den Programmen entfernt. Stattdessen wurden die Eingabedaten als konstante, globale Arrays im C-Quelltext angegeben und die Ergebnisdaten in einen weiteren Speicherbereich geschrieben. Ein weiteres Problem für die WCET-Analyse ist die dynamische Speicherverwaltung mittels `malloc`, `calloc`, etc. Deshalb wurden alle dynamisch allokierten Objekte durch Objekte mit einem statisch reservierten Speicherbereich ersetzt. In einigen Fällen waren dazu Worst-Case Annahmen bezüglich des maximal erforderlichen Speicherplatzes notwendig.

Nachdem die Benchmarks für die WCET-Analyse vorbereitet sind, müssen im nächsten Schritt die benötigten Benutzerangaben für eine erfolgreiche Analyse mit

aiT erstellt werden. Hauptsächlich geht es dabei um das Bestimmen der maximalen Iterationsanzahl der Schleifen, was insbesondere bei den komplexeren Benchmarks, die viele Schleifen enthalten, sehr aufwendig sein kann.

Um diese notwendigen Vorbereitungen für die WCET-Analyse von Benchmarks zu vermeiden, stellt die Mälardalen WCET research group eine Sammlung von Benchmarks zur Verfügung, die bereits für die WCET-Analyse vorbereitet sind [Mär06]. Aus dieser Sammlung wurden ebenfalls zwei Benchmarks verwendet.

Die folgende Liste enthält alle in dieser Arbeit verwendeten Benchmark Programme sowie ein kurze Beschreibung:

- **ADPCM**: *Adaptive Differential Pulse Code Modulation* ist ein einfaches Verfahren zur Sprachkompression. Es werden 16-bit PCM Samples in 4-bit Samples konvertiert. Es wird sowohl die Kodierung als auch die Dekodierung von jeweils 100 Samples untersucht.
- **Bubblesort**: Eine einfache Implementierung des Bubblesort Sortieralgorithmus auf einem Array mit 100 zufälligen Integer-Werten. Ein Herausforderung für die WCET-Analyse ist die Abhängigkeit der Iterationsanzahl der inneren Schleifen von der äußeren Schleife.
- **Cavity**: Eine medizinische Bildverarbeitungsanwendung, welche die Erkennung von Gehirntumoren unterstützt.
- **CRC**: CRC Prüfsummenberechnung für eine 40 Byte und eine 42 Byte Eingabe.
- **EPIC**: EPIC steht für *Efficient Pyramid Image Coder* und ist ein experimentelles Bildkompressionsverfahren. Aufgrund der Komplexität der Anwendung wurde lediglich der erste Schritt (der Aufruf der Funktion `build_pyr`) der Kodierung betrachtet. Es sind sehr viele Schleifen enthalten, deren Iterationsanzahlen stark variieren, wodurch eine exakte WCET-Analyse erschwert wird.
- **FIR**: Ein *finite impulse response* Filter angewendet auf einem Eingabearray mit 700 32-bit Werten.
- **G.721**: Die Referenzimplementierung des CCITT G.721 Audiokodierungverfahrens. Es wird die Kodierung und Dekodierung einer Audiosequenz mit 3200 16-bit PCM Samples untersucht.
- **GSM**: Eine Implementierung des in Mobilfunknetzen genutzten GSM Sprachkodierungsverfahrens. Es wird hier nur die Kodierung betrachtet. Als Eingabe dient die auch für G.721 verwendete Audiosequenz.
- **Motion Estimation (ME)**: Ein MPEG 4 *full search motion estimation* Algorithmus. Dieser Benchmark besteht aus einer Funktion, die eine sechsfach verschachtelte Schleife enthält.

- **MPEG2**: Eine Implementierung des MPEG 2 Videokompressionsverfahrens. Es wird nur der Motion Estimation Teil betrachtet. Aufgrund einiger schwer annotierbarer Schleifen sind die Abweichungen der WCET-Abschätzung von der gemessen Laufzeit für diesen Benchmark sehr hoch.
- **QSDPCM**: Ein *Quadtree Structured Difference Pulse Code Modulation* Algorithmus für die Videokomprimierung. Dieser Benchmark enthält eine siebenfach verschachtelte Schleife und hat eine ähnliche Struktur wie der Motion Estimation Benchmark.
- **SHA**: Eine Implementierung des NIST *Secure Hash Algorithm*, der für beliebige Eingabedaten einen 160 Bit Hash-Wert berechnet. Es wird ein Text bestehend aus 311.824 Zeichen als Eingabe verwendet.

Tabelle 2.1 enthält eine Übersicht der verwendeten Benchmarks sowie deren Anzahl an Quelltextzeilen, Schleifen und Funktionen.

Benchmark	**Quelltextzeilen**	**Anzahl der Funktionen**	**Anzahl der Schleifen**
ADPCM Coder	113	2	1
ADPCM Decoder	93	2	1
Bubblesort	22	1	2
Cavity	234	3	2
CRC	72	3	3
EPIC	334	6	41
FIR	52	2	2
G.721 Coder	380	12	8
G.721 Decoder	375	12	7
GSM	1066	36	48
Motion Estimation	82	1	8
MPEG2	1226	14	33
QSDPCM	57	1	7
SHA	140	6	9

Tabelle 2.1: Übersicht der verwendeten Benchmarks

3 Klassifizierung von Optimierungen

Aus dem in Abschnitt 2.3.1 beschriebenen Ablauf der WCET-Analyse mit aiT lassen sich verschiedene Faktoren ermitteln, die einen Einfluss auf die berechnete WCET-Abschätzung haben. Jede der Analyse-Phasen basiert auf unterschiedlichen Informationen über das Programm, die teilweise in vorangegangenen Phasen erzeugt wurden, direkt aus dem Programm gewonnen werden oder durch Benutzerangaben geliefert werden und einen großen Einfluss auf die WCET-Abschätzung haben.

Durch genaue Kenntnisse über die Faktoren, die einen Einfluss auf die Berechnungen der verschiedenen Analyse-Phasen haben, wird es möglich ein Programm dahingehend zu optimieren, dass die bestimmte WCET verringert wird. Das kann einerseits durch eine Verbesserung der Analysierbarkeit erfolgen, wodurch der Fehler der Abschätzung verringert wird, oder aber durch die Reduzierung der Anzahl der benötigten Taktzyklen auf dem WC Pfad.

Das Ziel dieser Arbeit ist es nun, den Einfluss bekannter Compiler-Optimierungen, wie sie beispielsweise in [BGS94] oder [Muc97] beschrieben werden, und meist die Reduzierung der ACET oder der Codegröße zum Ziel haben, auf die WCET (bzw. auf die WCET-Abschätzung) zu untersuchen. In einem erstem Schritt wurde dazu eine Klassifizierung solcher Optimierungen bezüglich des möglichen Einflusses auf verschiedene, für die WCET-Analyse bedeutsame Faktoren, durchgeführt.

Anhand dieser theoretischen Klassifizierung wurden anschließend einige erfolgversprechende Optimierungen für eine genauere Untersuchung ausgewählt. Die Ergebnisse dieser Untersuchungen werden in den Kapiteln 4 und 5 vorgestellt.

3.1 Loop Bounds

Bei der WCET-Analyse werden obere Schranken für die Iterationsanzahl sämtlicher Schleifen benötigt. Diese maximale Iterationsanzahl einer Schleife hat einen großen Einfluss auf die WCET, da sie bestimmt, wie oft die WCET des Schleifenkörpers in die gesamte WCET eingeht. Schleifen machen meist einen Großteil der gesamten Programmlaufzeit aus. Durch ungenaue Loop Bounds werden auf dem WC Pfad mehr Schleifeniterationen berücksichtigt, als bei einer realen Ausführung möglich sind. Dadurch kann es schnell zu hohen Fehlern bei der WCET-Abschätzung kommen.

Besonders problematisch sind Schleifen, deren Iterationsanzahl kontextabhängig

stark variiert. Dies sind beispielsweise Schleifen deren Iterationsbereich abhängig ist von einem Funktionsparameter oder von der Indexvariablen einer äußeren Schleife. Der erste Fall tritt z.B. bei den C-Funktionen `memset` oder `memcpy` auf, denen als Parameter die Anzahl der zu setzenden/kopierenden Bytes übergeben wird. Ein Beispiel für den zweiten Fall sind zwei verschachtelte Schleife mit einem dreieckigen Iterationsbereich:

```
for (i=0; i<n; i++) {
  for (j=0; j<=i; i++) {
    ...
  }
}
```

Beide Schleife haben maximal n Iterationen. Entsprechend müssten die Loop Bounds für die WCET-Analyse angegeben werden, was dazu führt, dass n^2 Ausführungen der Instruktionen in der inneren Schleife gezählt werden, statt der tatsächlichen $\frac{1}{2}n(n+1)$.

In aiT werden die Loop Bounds für jeden betrachteten Kontext separat bestimmt. Dadurch können diese Probleme teilweise vermieden werden. Dies allerdings nur, wenn genügend Kontexte unterschieden werden, und wenn die Loop Bounds auch automatisch bestimmt werden können. Bei Schleifen werden jedoch üblicherweise nur die ersten Iterationen in einem eigenen Kontext betrachtet. Alle weiteren Iterationen werden in einem Kontext zusammengefasst und haben somit gemeinsame Loop Bounds für innere Schleifen. Durch den Benutzer angegebene Loop Bounds sind in aiT immer für alle Kontexte gültig und überschreiben automatisch bestimmte Loop Bounds einzelner Kontexte.

Verschiedene Compiler-Optimierungen könnten zu einer Verbesserung der Probleme durch ungenaue Loop Bounds beitragen. Zum einem durch Transformationen, die die automatische Loop Bound Analyse erleichtern oder durch Umstrukturierungen, die eine exakte Angabe der Loops Bounds ermöglichen.

- **Loop Normalization**
  Diese Optimierung transformiert Schleifen derart, dass die Indexvariable zu Beginn den Wert 0 hat und mit jeder Iteration um 1 inkrementiert wird. Diese normalisierte Form könnte bei der Loop Bound Analyse von Vorteil sein.

- **Loop Reversal**
  Loop Reversal ändert die Richtung, in der der Iterationsbereich einer Schleife durchlaufen wird. Läuft der Iterationsbereich einer Schleife von einem positiven Wert n bis 0, so wird bei einigen Architekturen keine Vergleichsoperation in der Schleife benötigt, sondern lediglich eine bedingte Sprunginstruktion. Auch diese Optimierung könnte sich auf die Loop Bound Analyse auswirken, was allerdings stark von der Architektur und dem Vorgehen der Loop Bound Analyse abhängig ist.

- **Loop Coalescing**
  Beim *Loop Coalescing* werden mehrere verschachtelte Schleifen durch eine einfache Schleife ersetzt. Die Werte aller ursprünglichen Indexvariablen werden aus der Indexvariablen der neuen Schleife berechnet (siehe Abbildung 3.1). Nach dieser Transformation müssen bei der WCET-Analyse nur noch Loop Bound Annotationen für eine Schleife angegeben werden. Die Probleme bei inneren Schleifen, deren Loop Bounds von einer äußeren Schleife abhängen, könnten somit vermieden werden. Für das zu Beginn des Abschnitts vorgestellte Beispiel verschachtelter Schleifen mit dreieckigem Iterationsbereich könnte für die neue Schleife die exakte Iterationsanzahl $\frac{1}{2}n(n+1)$ angegeben werden. Es muss allerdings auch der entstehende Aufwand zur Berechnung der ursprünglichen Indexvariablen berücksichtigt werden.

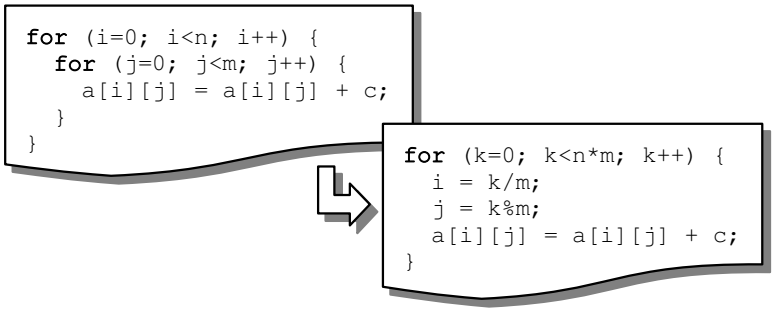

Abbildung 3.1: Beispiel Loop Coalescing

- **Loop Skewing**
  Eine Transformation, die negative Auswirkungen für die WCET-Analyse haben könnte, ist das *Loop Skewing*. *Loop Skewing* wird üblicherweise im Kombination mit *Loop Interchange* durchgeführt, bei dem die innere und äußere Schleife vertauscht werden. Durch diese Transformationen kann bei „*wavefront*“ Berechnungen auf Arrays die Parallelisierbarkeit verbessert werden. Dazu wird der ursprünglich rechteckige Iterationsbereich verzerrt, so dass ein trapezförmiger Iterationsbereich entsteht. Dabei entsteht eine innere Schleife mit variablen Loop Bounds, abhängig von der äußeren Schleife. Eine exakte Angabe der Loop Bounds ist somit nicht mehr möglich.

  Abbildung 3.2 zeigt ein Beispiel für das *Loop Skewing* mit anschließendem *Loop Interchange*. Vor den Transformationen beträgt die Iterationsanzahl beider Schleifen n. Nach der Transformation hat die äußere Schleife $2n-1$ Iterationen und die Iterationsanzahl der inneren Schleife liegt abhängig von der äußeren Schleife zwischen 1 und n. Bei einer Angabe von n als maximale Iterationsanzahl würde die WCET-Analyse somit $(2n-1)n$ Ausführungen des Schleifenkörpers zählen, anstelle von n^2 Ausführungen vor dem *Loop Skewing*.

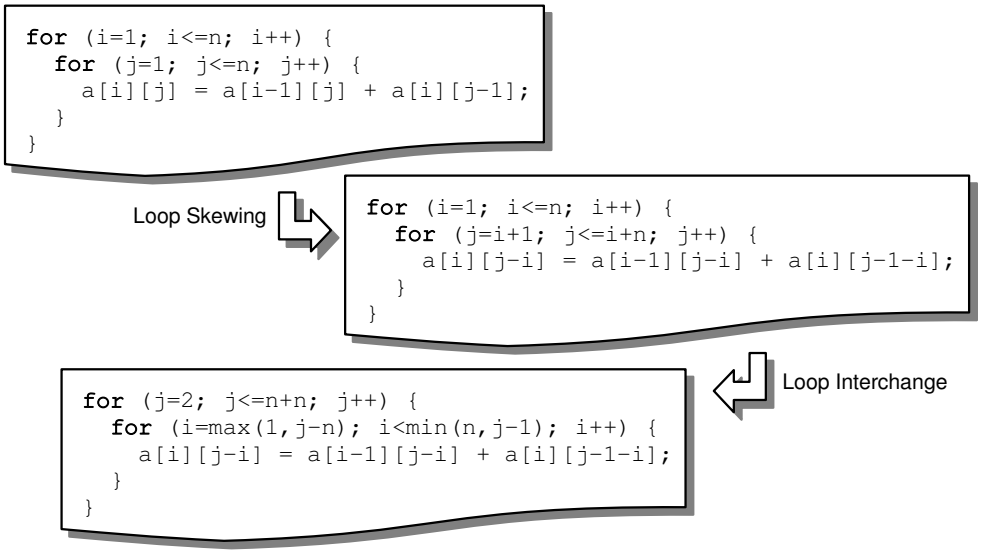

Abbildung 3.2: Beispiel Loop Skewing

- **Procedure Cloning**
  Beim *Procedure Cloning* (auch *Function Specialization*) werden Kopien von Funktionen erzeugt, bei denen bestimmte Parameter durch Konstanten ersetzt werden. Ein Vorteil für die WCET-Analyse ist, dass nun für jede Kopie die Loop Bounds separat angegeben werden können. Besonders bei Schleifen, deren Loop Bounds abhängig sind von den durch Konstanten ersetzten Parametern, könnte das zu einer erheblichen Reduzierung des Fehlers bei der WCET-Abschätzung führen.

3.2 Ergebnisse der Value Analyse

Die Ergebnisse der Value Analyse haben einen großen Einfluss auf die Genauigkeit der WCET-Abschätzung. Bei der Cache Analyse werden die ermittelten Adressen von Speicherzugriffen benötigt. Können diese Adressen nicht bestimmt werden, so muss vom schlimmsten Fall ausgegangen werden (i.d.R. *cache misses*). Die Pipeline Analyse müsste bei einem Speicherzugriff mit unbekannter Adresse von einem Zugriff auf den langsamsten Speicherbereich des Systems ausgehen. Die ermittelten Registerwerte werden auch verwendet, um Sprungbedingungen auszuwerten und so *infeasible paths* zu identifizieren. Kann eine Sprungbedingung nicht ausgewertet werden, so müssen beide möglichen Pfade betrachtet werden, was zum einem den Fehler der WCET-Abschätzung erhöht, und zum anderen auch Auswirkungen auf die Analysezeit haben kann, da für die Basisblöcke beider Pfade die WCET berechnet werden muss und auch die Komplexität der Pfadanalyse mit der Anzahl der möglichen Pfade steigt.

Es lassen sich eine Reihe verschiedener Ursachen nennen, weshalb Registerwerte nicht genau bestimmt werden können:

- Der Wert eines Registers ist abhängig von Eingabedaten, die während der statischen Analyse unbekannt sind.
- Ein Register hat im betrachteten Kontext verschiedene Werte, beispielsweise wenn nicht jeder Aufruf einer Funktion (bzw. jede Iteration einer Schleife) durch einen eigenen Kontext repräsentiert wird. In diesem Fall kann höchstens ein Intervall möglicher Werte ermittelt werden.
- Der Wert eines Registers wird in einem *if*-Block verändert, dessen Bedingung nicht statisch ausgewertet werden kann.
- Der Wert eines Registers wurde in den Speicher ausgelagert, und durch einen Schreibzugriff auf eine unbekannte Adresse mussten sämtliche Informationen über Speicherinhalte verworfen werden.

Um die WCET-Überschätzung durch unbekannte Registerwerte zu reduzieren, könnten Transformationen durchgeführt werden, die der Value Analyse das Ermitteln der Werte bestimmter Variablen, die einen großen Einfluss auf die WCET-Abschätzung haben, ermöglichen. Außerden könnten bereits durch den Compiler Adressen von Speicherzugriffen bestimmt und an die WCET-Analyse weitergegeben werden. Die folgenden Compiler-Optimierungen lassen sich diesem Bereich zuordnen:

- **Alias Analyse**
  Bei der Alias Analyse wird bestimmt, welche Referenzen auf dieselbe Speicherzelle verweisen. Die gewonnen Informationen könnten in Form von Adressen von Speicherzugriffen an die WCET-Analyse weitergegeben werden.
- **Common-Subexpression Elimination (CSE)**
  Bei dieser Optimierung wird die Berechnung eines identischen Teilausdruckes an mehreren Stellen im Programm durch eine einzige Berechnung ersetzt, deren Ergebnis in einer Variablen zwischengespeichert und an den anderen Stellen genutzt wird. Neben den positiven Auswirkungen durch weniger Instruktionen und die nur einmalige Berechnung des Wertes bei der Value Analyse sind auch negative Auswirkungen denkbar, wenn beispielsweise die neu eingeführte Variable in den Speicher ausgelagert wird.
- **Loop Invariant Code Motion (LICM)**
  Durch LICM werden Berechnungen innerhalb von Schleifen, deren Ergebnis nicht von den Indexvariablen abhängt, vor die Schleife verschoben. Ähnlich wie bei der CSE sind auch hier positive und negative Auswirkungen auf die WCET denkbar.

3.3 Cache Verhalten

Die Cache Analyse in aiT versucht durch abstrakte Interpretation, Speicherblöcke zu ermitteln, die sich zu einem bestimmten Programmpunkt im Cache befinden müssen. So können einige Speicherzugriffe als sicherer *cache hit* erkannt werden. Die Analyse ist im hohen Maße abhängig von den durch die Value Analyse ermittelten Adressen von Speicherzugriffen. Ein einziger Speicherzugriff auf eine unbekannte Adresse kann zum Verlust sämtlicher Informationen über den Cache-Zustand führen. Bei Schleifen, die große Arrays bearbeiten, können durch die begrenzte Anzahl betrachteter Kontexte meist nur wenige Zugriffe in den ersten Iterationen genau bestimmt werden. Die meisten Zugriffe müssten aber als *cache misses* betrachtet werden. Durch diese Probleme wirken sich Verbesserungen des Datencache Verhaltens durch verschiedene Schleifentransformationen oft nicht im gleichen Maße auf die WCET aus, wie auf die ACET.

Bei der Analyse eines Instruktionscaches treten diese Probleme nicht auf, da die Adressen der geladenen Instruktionen bekannt sind. Durch geeignete Optimierungen könnte versucht werden, die Anzahl der *cache misses* auf dem WC Pfad zu reduzieren.

Folgende Optimierungen können zur Verbesserung des Cache-Verhaltens beitragen:

- **Loop Distribution**
  Bei dieser Optimierung wird eine Schleife in mehrere Schleifen mit identischem Iterationsbereich aufgeteilt, die jeweils einen Teil der Instruktionen der originalen Schleife enthalten. So kann die Größe der Schleifenkörper an die Größe eines Instruktionscaches angepasst werden.

- **Loop Tiling**
  Beim *Loop Tiling* wird der Iterationsraum von Schleifen in kleinere Blöcke unterteilt, so dass die verwendeten Daten eines Blockes in den Daten-Cache passen. So kann die Anzahl der *cache misses* reduziert werden.

3.4 Pipeline Verhalten

Bei Prozessoren, die eine Pipeline verwenden, kann es in verschiedenen Situationen notwendig sein, die Pipeline-Verarbeitung für einige Taktzyklen anzuhalten (*pipeline stall*). Dies ist z.B. erforderlich bei Speicherzugriffen, die mehrere Taktzyklen benötigen, oder bei sog. *pipeline hazards*. Man unterscheidet Struktur-Hazards, Daten-Hazards und Kontroll-Hazards [HP96]. Struktur-Hazards treten auf, wenn eine Hardwarekomponente des Prozessors von verschiedenen Pipelinestufen gleichzeitig benötigt wird. Daten-Hazards können auftreten, wenn die Ausführung einer

Instruktion abhängig ist vom Ergebnis vorangegangener Instruktionen. Kontroll-Hazards entstehen bei Instruktionen, die den *program counter (PC)* verändern, wie z.B. Sprünge oder Funktionsaufrufe.

Durch die einfache 3-stufige Pipeline des ARM7TDMI (siehe Kapitel 2.4.1) kommen bei diesem Prozessor lediglich Kontroll-Hazards vor. Jeder Kontrollflusstransfer führt zu einem *pipeline stall* von 2 Taktzyklen, da die Instruktionen in den ersten beiden Pipeline-Stufen verworfen werden müssen.

Die Pipeline-Verarbeitung des Infineon TC1796 ist etwas komplexer (siehe Kapitel 2.4.2). Hier sind alle drei Arten von Hazards möglich. Um die Performance-Verluste durch Kontroll-Hazards zu verringern, unterstützt der TC1796 eine statische Sprungvorhersage. Die Länge der *pipeline stalls* verursacht durch Sprunginstruktionen ist abhängig von der Art des Sprunges und der Korrektheit der Vorhersage. Sie beträgt maximal zwei Taktzyklen.

Die Auswirkungen von *pipeline stalls* auf die WCET bzw. die WCET-Analyse entsprechen im Wesentlichen den Auswirkungen auf die durchschnittliche Laufzeit. Bei der Pipeline Analyse wird das Pipeline-Verhalten des entsprechenden Prozessors exakt simuliert und *pipeline stalls* werden berücksichtigt, genau wie sie bei einer realen Ausführung auftreten würden. Durch fehlende Informationen während der Simulation sind allerdings häufig Worst-Case Annahmen notwendig, wodurch mehr *pipeline stalls* gezählt werden als bei einer realen Ausführung vorkommen würden.

Durch verschiedene Compiler-Optimierungen können die negativen Auswirkungen von *pipeline stalls* verringert werden. Kontroll-Hazards können vermieden werden, indem die Anzahl der Sprünge reduziert wird. Struktur- und Daten-Hazards können durch eine möglichst günstige Anordnung der Instruktionen verringert werden. Die bisher verwendeten Optimierungen haben oft die Reduzierung der *pipeline stalls* auf dem am häufigsten ausgeführten Pfad zum Ziel, der durch *profiling* ermittelt wird. Zur Verringerung der WCET könnten diese Optimierungen so angepasst werden, dass stattdessen der WC Pfad optimiert wird.

- **Code Positioning**
  Beim *Code Positioning* wird eine Anordnung der Basisblöcke eines Programms gesucht, die zu einer Minimierung der Sprünge auf einem häufig ausgeführten Pfad führt. Zur Reduzierung der WCET kann stattdessen der WC Pfad betrachtet werden.

- **Loop Unrolling**
  Durch *Loop Unrolling* wird die Anzahl der Iterationen einer Schleife reduziert, indem der Schleifenkörper in einer Iteration mehrfach ausgeführt wird. Damit wird die Anzahl der Sprünge, und damit die der Kontroll-Hazards, reduziert. Bei einer nicht konstanten Anzahl an Iterationen ist allerdings eine zusätzliche Schleife zur Behandlung der letzten Iterationen notwendig, die negative Auswirkungen auf die WCET-Abschätzung haben könnte.

- **Loop Nest Splitting**
  Loop Nest Splitting ist eine Optimierung, die durch eine Teilung des Iterationsraumes mehrfach verschachtelter Schleifen die Anzahl der ausgeführten *if*-Anweisungen in der innersten Schleife reduziert. Auf diese Weise wird auch die Anzahl der ausgeführten Sprünge reduziert.

- **Instruction Scheduling**
  Das *Instruction Scheduling* wird bei der Codeerzeugung im Backend eines Compilers durchgeführt. Durch eine geschickte Anordnung der Instruktionen können *pipeline stalls* vermieden werden.

- **Scalar Replacement**
  Beim *Scalar Replacement* wird ein Array-Element, auf das häufig zugegriffen wird, dessen Wert aber nicht verändert wird, in eine skalare Variable geladen, die möglichst in einem Register gespeichert wird. So kann die Anzahl der Speicherzugriffe reduziert werden.

3.5 Kontrollfluss

In der letzten Phase der WCET-Analyse wird ein Ausführungspfad ermittelt, der zu einer maximalen Laufzeit führt. Dabei werden Einschränkungen bezüglich des möglichen Kontrollflusses berücksichtigt, die in vorherigen Analysephasen ermittelt wurden oder aus Benutzerangaben stammen. Die Genauigkeit der WCET-Abschätzung hängt beträchtlich von diesen Informationen über mögliche Ausführungspfade ab.

Einige Kontrollflussstrukturen sind bei der WCET-Analyse nur schwer handhabbar und können zur Bestimmung eines WCET-Pfades führen, der bei einer realen Ausführung gar nicht vorkommen kann. Entsprechend erhöht sich dadurch der Fehler der WCET-Abschätzung. Dazu kann auch die Begrenzung der Anzahl an Schleifeniterationen gezählt werden, die häufig nicht exakt möglich ist und so zur Berücksichtigung zu vieler Iterationen führt (vgl. Abschnitt 3.1). Problematisch sind auch unstrukturierte Schleifen, die mehrere Eintritts- oder Austrittspunkte besitzen. Die Schleife im folgenden Beispiel besitzt durch die *return*-Anweisung in der Schleife einen zweiten Austrittspunkt:

```
int foo() {
  for (i=0; i<n; i++) {
    if ( cond ) return expr1;
    ...
  }

  return expr2;
}
```

Die Berechnung von `expr1` wird höchstens einmal, beim Verlassen der Schleife, ausgeführt, und wird bei der Schleifentransformation von aiT deshalb nicht innerhalb

der rekursiven Funktion, die die Schleife repräsentiert, platziert, sondern nach dem Aufruf dieser Funktion in der Funktion, die die Schleife enthält (siehe Abbildung 3.3). Da bei der Pfadanalyse in aiT der Pfad innerhalb der rekursiven Funktion unabhängig vom Pfad nach dem Funktionsaufruf gewählt wird, kann es vorkommen, dass die Berechnung von `expr2` auf dem WC Pfad liegt, obwohl die Schleife über die *return*-Anweisung verlassen wird.

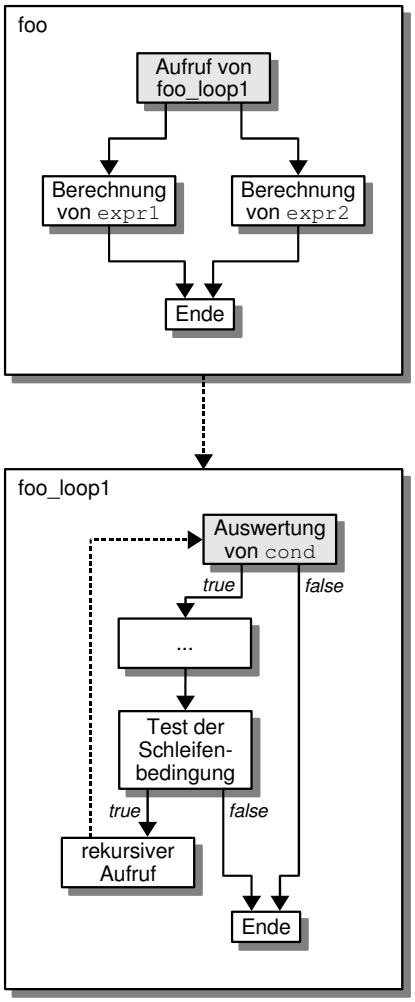

Abbildung 3.3: Schleife mit mehreren Austrittspunkten

Schleifen mit mehreren Eintrittspunkten werden von aiT gar nicht als Schleifen erkannt und somit auch nicht in rekursive Funktionen umgewandelt. Hier ist eine Begrenzung der Iterationsanzahl durch Benutzerangaben zwingend erforderlich.

Ein weiterer Fall, der zu einer Überschätzungen der WCET beitragen kann, sind aufeinanderfolgende *if*-Anweisungen mit sich gegenseitig ausschließenden Bedingungen:

```
if (x<10) {
  block1;
}

block2; // x wird hier nicht verändert

if (x>=10) {
  block3;
}
```

In diesem Beispiel werden `block1` und `block3` niemals gemeinsam ausgeführt. Falls der Wert von `x` allerdings nicht durch die Value Analyse ermittelt werden kann, würden sowohl `block1` als auch `block3` auf dem ermittelten WC Pfad liegen.

Um den Fehler bei der WCET-Abschätzung, der durch schwer analysierbare Kontrollflussstrukturen entsteht, zu verringern, könnten Compiler-Optimierungen eingesetzt werden, die für die WCET-Analyse günstigere Strukturen erzeugen oder aber zusätzliche Informationen über mögliche Kontrollflusspfade bereitstellen, die bei der WCET-Analyse berücksichtigt werden können. Die folgenden bewährten Optimierungen könnten dazu geeignet sein:

- **Inlining**
  Beim *Inlining* werden Funktionsaufrufe ersetzt durch den gesamten Inhalt der aufzurufenden Funktion. Bei häufig aufgerufenen Funktionen kann so der durch die Funktionsaufrufe entstehende Overhead vermieden werden. Für die WCET-Analyse könnte zusätzlich von Vorteil sein, dass nach dem *Inlining* einer Funktionen speziell angepasste Benutzerangaben für jede einkopierte Version der Funktion möglich sind.

- **Procedure Cloning**
  Ähnliche Vorteile könnten durch die Optimierung *Procedure Cloning* erreicht werden, die bereits in Abschnitt 3.1 vorgestellt wurde. Außer den Loop Bounds können auch andere Kontrollflussinformationen, speziell an die verschiedenen Kopien der Funktion angepasst, angegeben werden.

- **Loop Nest Splitting**
  Auch das bereits in Abschnitt 3.4 vorgestellte *Loop Nest Splitting* kann zu einer Verringerung der WCET-Überschätzung führen, wenn die während der Optimierung gewonnen Informationen über die Ausführungsanzahl der verschiedenen Programmteile bei der Pfadanalyse berücksichtigt werden.

3.6 Auswahl von Optimierungen

Ziel der in diesem Kapitel vorgestellten Klassifizierung von Compiler-Optimierungen bezüglich möglicher Auswirkungen auf verschiedene Einflussfaktoren der WCET-

	Loop Bounds	Ergebnisse der Value Analyse	Cache Verhalten	Pipeline Verhalten	Kontroll-fluss
Loop Normalization	•				
Loop Reversal	•				
Loop Coalescing	•				
Loop Skewing	•				
Procedure Cloning	•				•
Alias Analysis		•			
CSE		•			
LICM		•			
Loop Distribution			•		
Loop Tiling			•		
Code Positioning				•	
Loop Unrolling				•	
Loop Nest Splitting				•	•
Instruction Scheduling				•	
Scalar Replacement				•	
Inlining					•

Tabelle 3.1: Klassifizierung von Optimierungen

Analyse war es, Optimierungen zu ermitteln, die für die WCET-Optimierung besonders erfolgversprechend scheinen, um diese anschließend genauer zu untersuchen. In Tabelle 3.1 ist die hier vorgestellte Zuordnung von Optimierungen zu den identifizierten Einflussfaktoren noch einmal zusammengefasst dargestellt.

Häufig ist bei der statischen WCET-Analyse die Überschätzung recht hoch. Die Ursachen hierfür sind oft schwierig zu analysierende Kontrollflussstrukturen oder ungenaue Werte für die maximale Iterationsanzahl von Schleifen. Durch Optimierungen, die den Kontrollfluss vereinfachen oder die Angabe der maximalen Iterationsanzahl verbessern, könnte die Überschätzung reduziert werden, wodurch sich möglicherweise Verbesserungen der WCET-Abschätzung ergeben, die über den Verbesserungen der ACET nach den entsprechenden Optimierungen liegen. Die Optimierung *Procedure Cloning* kann sowohl die Loop Bounds als auch den Kontrollfluss beeinflussen und wurde deshalb für eine genauere Untersuchung ausgewählt. Eine weitere ausgewählte Optimierung ist das *Loop Nest Splitting*, welches die Kontroll-Struktur verändert und auch die Anzahl der *pipeline stalls* reduziert.

Desweiteren wurden die Optimierungen *Loop Unrolling* und *Code Positioning* ausgewählt, die ebenfalls das Pipeline Verhalten beeinflussen. Durch diese Optimierun-

gen könnte eine Reduzierung der WCET erreicht werden, indem sie speziell zur Vermeidung von *pipeline stalls* auf dem Worst-Case Pfad eingesetzt werden. Dazu müssen bei der Durchführung der Optimierungen die Ergebnisse einer vorherigen WCET-Analyse berücksichtigt werden.

4 Kontrollfluss-Optimierungen

Anhand der in Kapitel 3 durchgeführten Klassifizierung und ersten Bewertung von Compileroptimierungen bezüglich des Einflusses auf die Worst-Case Laufzeit wurden einige der Optimierungen für eine genauere Untersuchung ausgewählt. In diesem Kapitel werden die Ergebnisse der Untersuchung der drei Kontrollfluss-Transformationen Loop Nest Splitting, Procedure Cloning und Loop Unrolling vorgestellt.

Die Optimierungen wurden auf dem C-Quelltext verschiedener Benchmark-Programme durchgeführt, um anschließend die Auswirkungen auf die WCET, die ACET und die Codegröße zu analysieren. Dabei wurde auch nach Möglichkeiten gesucht, die WCET-Abschätzung durch zusätzliche Benutzerangaben zu verbessern, die durch die Transformationen ermöglicht werden.

4.1 Loop Nest Splitting

Die erste untersuchte Optimierung ist das in [FM03, FM04] vorgestellte Loop Nest Splitting. Durch diese Optimierung kann die Anzahl der ausgeführten *if*-Anweisungen innerhalb mehrfach verschachtelter Schleifen reduziert werden. Insbesondere bei Multimedia Anwendungen werden häufig große Datenmengen mit Hilfe mehrfach verschachtelter Schleifen verarbeitet. Oft befindet sich der eigentliche Algorithmus dabei in der innersten Schleife, die auch einige *if*-Anweisungen zur separaten Behandlung von Sonderfällen, wie beispielsweise die Randbereiche bei der Bildverarbeitung, enthält. Abbildung 4.1 zeigt einen MPEG 4 Motion Estimation Algorithmus als Beispiel für eine solche Anwendung.

Die *if*-Anweisungen führen zu einem sehr unregelmäßigem Kontrollfluss in der innersten Schleife, der negative Auswirkungen sowohl auf die durchschnittliche, als auch auf die Worst-Case Laufzeit hat. In jeder Iteration müssen die Bedingungen ausgewertet werden und ggf. Sprünge durchgeführt werden, die Kontroll-Hazards, und somit *pipeline stalls*, zur Folge haben.

Für die WCET-Analyse ergeben sich aus einer solchen Struktur weitere Nachteile. Für eine genaue WCET-Abschätzung müssten die Bedingungen der *if*-Anweisungen ausgewertet werden, da ohne diese Informationen immer der längere Pfad gewählt werden muss. Bei *if*-Anweisungen innerhalb verschachtelter Schleifen, deren Bedingungen von den Schleifenvariablen abhängen, ist dies aber nur selten möglich. Durch die tief verschachtelten Schleifen muss die Anzahl der unterschiedenen Kontexte für Schleifen meist sehr gering gewählt werden, um vertretbare Analysezeiten zu er-

```
for (x=0; x<36; x++) {
  x1=4*x;
  for (y=0; y<49; y++) {
    y1=4*y;
    for (k=0; k<9; k++) {
      x2=x1+k-4;
      for (l=0; l<9; l++) {
        y2=y1+l-4;
        for (i=0; i<4; i++) {
          x3=x1+i; x4=x2+i;
          for (j=0; j<4; j++) {
            y3=y1+j; y4=y2+j;
            if (x3<0 || 35<x3 || y3<0 || 48<y3)
              then_block_1; else else_block_1;
            if (x4<0 || 35<x4 || y4<0 || 48<y4)
              then_block_2; else else_block_2; }}}}}}
```

Abbildung 4.1: MPEG 4 Motion Estimation Algorithmus

reichen. Dadurch kann die Value Analyse die Werte der Schleifenvariablen nur für einige wenige Iterationen bestimmen. Dazu kommt als weiteres Problem, dass die Schleifenvariablen der äußeren Schleifen aufgrund der begrenzten Anzahl an Prozessorregistern, oft in den Speicher ausgelagert werden. Durch einen schreibenden Speicherzugriff auf eine unbekannte Adresse gehen die durch die Value Analyse ermittelten Werte somit verloren. Da solche Anwendungen üblicherweise Datenarrays manipulieren, lassen sich Schreibzugriffe auf unbekannte Adressen aber meist nicht vermeiden.

Ein weiterer Fehler bei der WCET-Abschätzung entsteht durch die Art der Auswertung langer Bedingungen, die aus mehreren, durch logische UND/ODER Operatoren verknüpften, Ausdrücken bestehen. Folgt aus der Auswertung eines Teilausdruckes, dass die gesamte Bedingung erfüllt bzw. nicht erfüllt ist, so werden die weiteren Teilausdrücke nicht mehr ausgewertet. Dadurch folgt auf die Instruktionen zur Auswertung eines Teilausdruckes jeweils eine Sprunginstruktion. Aus den oben genannten Gründen können die Sprungbedingungen nur selten durch die Value Analyse ausgewertet werden, so dass für die meisten Iterationen die Auswertung aller Teilausdrücke auf dem ermittelten Worst-Case Pfad liegt.

Um die Anzahl der ausgeführten *if*-Anweisungen zu reduzieren, werden bei der Durchführung der Optimierung Iterationen bestimmt, in denen alle *if*-Anweisungen nachweislich erfüllt sind. Anschließend wird der Iterationsraum der verschachtelten Schleifen so aufgeteilt, dass für einen möglichst großen Teil der Iterationen die *if*-Anweisungen entfallen können. Zusätzlich werden auch Bedingungen erkannt, die keinen Einfluss auf den Kontrollfluss haben und somit aus dem Programmcode entfernt werden können.

Für das Beispiel aus Abbildung 4.1 kann so ermittelt werden, dass die Bedingungen $\mathtt{x3} < 0$ und $\mathtt{y3} < 0$ niemals erfüllt sind, und dass die beiden *if*-Anweisungen erfüllt sind, falls $\mathtt{x} \geq 10$ oder $\mathtt{y} \geq 14$. Damit ergibt sich die optimierte Version des Beispiels wie in Abbildung 4.2 dargestellt. In der y Schleife wurde eine zusätzliche *if*-Anweisung eingefügt. Deren *then*-Teil enthält die vier inneren Schleifen des Programms, wobei die innere Schleife keine *if*-Anweisungen mehr enthält, sondern lediglich die *then*-Pfade. Der *else*-Teil der neu eingefügten *if*-Anweisung enthält die inneren Schleifen, genau wie im ursprünglichen Programm. Um die unnötige wiederholte Auswertung der zusätzlich eingefügten *if*-Anweisung zu vermeiden, wird im *then*-Teil eine weitere Schleife eingefügt, die das Inkrementieren der Variablen y bis zu ihrem Maximalwert übernimmt.

```
for (x=0; x<36; x++) {
  x1=4*x;
  for (y=0; y<49; y++)
    if (x>=10 || y>=14)
      for (; y<49; y++)
        for (k=0; k<9; k++) {
          x2=x1+k-4;
          for (l=0; l<9; l++) {
            y2=y1+l-4;
            for (i=0; i<4; i++) {
              x3=x1+i; x4=x2+i;
              for (j=0; j<4; j++) {
                then_block_1; then_block_2; }
    else { y1=4*y;
      for (k=0; k<9; k++) {
        x2=x1+k-4;
        for (l=0; l<9; l++) {
          y2=y1+l-4;
          for (i=0; i<4; i++) {
            x3=x1+i; x4=x2+i;
            for (j=0; j<4; j++) {
              y3=y1+j; y4=y2+j;
              if (0 || 35<x3 || 0 || 48<y3)
                then_block_1; else else_block_1;
              if (x4<0 || 35<x4 || y4<0 || 48<y4)
                then_block_2; else else_block_2; }}}}}}
```

Abbildung 4.2: MPEG 4 Motion Estimation Algorithmus nach Loop Nest Splitting

4.1.1 Ablauf der Optimierung

Die Optimierung Loop Nest Splitting basiert auf einer Darstellung des Iterationsraumes verschachtelter Schleifen und der Bedingungen von *if*-Anweisungen als Polyeder

bzw. Polytope. Zu Beginn wird der C-Quelltext untersucht, um Schleifen zu identifizieren, die für die Optimierung geeignet sind. Anschließend wird jede in den Schleifen vorkommende Bedingung durch ein Polytop dargestellt.

Im nächsten Schritt wird geprüft, ob Bedingungen vorkommen, die immer erfüllt bzw. nicht erfüllt sind (wie z.B. `x3` $<$ 0 und `y3` $<$ 0 in Abbildung 4.1). Solche Bedingungen werden in der *if*-Anweisung durch den entsprechenden Wahrheitswert ersetzt und werden bei der Analyse nicht weiter berücksichtigt.

Die Polytope der übrigen Bedingungen werden im nächsten Schritt durch optimierte Polytope ersetzt. Diese optimierten Polytope ergeben sich aus einer Beschränkung des Wertebereichs der Schleifenvariablen, so dass die zugehörige Bedingung immer erfüllt ist. Ein optimiertes Polytop ist somit eine Teilmenge des ursprünglichen Polytops. Es wird durch einen genetischen Algorithmus so gewählt, dass die Anzahl der ausgeführten *if*-Anweisungen, bei einer Aufteilung des Iterationsraumes entsprechend des Polytopes, minimiert wird.

Um einen Iterationsbereich zu bestimmen, in dem alle *if*-Anweisungen erfüllt sind, werden als nächstes die bisher einzeln betrachteten Bedingungen durch Schnitt bzw. Vereinigung der Polytope zusammengefasst. Es entsteht eine Vereinigung mehrerer Polytope, die jeweils einen Iterationsbereich darstellen, in dem alle *if*-Anweisungen erfüllt sind.

Als letzter Schritt wird durch einen weiteren genetischen Algorithmus eine Teilmenge dieser Polytope gewählt, wobei wieder die Anzahl der ausgeführten *if*-Anweisungen minimiert wird. Entsprechend dieser Auswahl wird die zusätzliche *if*-Anweisung zur Aufteilung des Iterationsraumes generiert.

4.1.2 Annotationen für die WCET-Analyse

Allein durch diese Code-Transformation kann die WCET allerdings nicht reduziert werden. Die neu eingefügte *if*-Anweisung kann, genau wie die anderen *if*-Anweisungen innerhalb der Schleifen, höchstens für einige Iterationen ausgewertet werden. Für alle anderen Iterationen müsste die Pfad Analyse den längeren der beiden Pfade wählen. Würde hier jeweils der *else*-Pfad gewählt, ergäbe sich eine WCET ähnlich der vor dem Loop Nest Splitting. Durch die zusätzliche *if*-Anweisung wäre der Wert sogar etwas höher. Tatsächlich ist bei der WCET-Analyse allerdings häufig der *then*-Pfad der längere, da es hier durch die zusätzlich eingefügte Schleife zu einer hohen Überschätzung der WCET kommen kann. Der Zusammenhang zwischen den beiden Schleifen, die dieselbe Variable inkrementieren, lässt sich durch einfache Loop Bound Annotationen nicht berücksichtigen.

Während der Optimierung kann jedoch leicht die exakte Ausführungsanzahl der verschiedenen Pfade berechnet werden. Das Ergebnis des Optimierungs-Algorithmus ist ein Polytop, das die Aufteilung des Iterationsraumes durch eine zusätzliche *if*-Anweisung beschreibt. Jeder Punkt innerhalb dieses Polytops repräsentiert eine Ite-

ration der verschachtelten Schleifen, für die die generierte *if*-Anweisung erfüllt ist. Indem die Anzahl der Punkte des Polytops ermittelt wird, kann also die Anzahl der Ausführungen des *then*-Pfades bestimmt werden. Auf ähnliche Weise lässt sich die Anzahl der Ausführungen des *else*-Pfades bestimmen. Diese Informationen können durch *flow*-Annotationen an aiT weitergegeben werden.

Für das Beispiel aus Abbildung 4.2 erhält man eine Ausführungszahl von 36 für den *then*-Pfad und 140 für den *else*-Pfad. Der geringere Wert beim *then*-Pfad ergibt sich durch die zweite y Schleife. Die Anzahl der Ausführungen des Schleifenkörpers der jeweils innersten Schleifen ist im *then*-Teil mit 2.104.704 viel höher als im *else*-Teil mit 181.440. Mit Hilfe dieser Werte werden folgende Annotationen für aiT erstellt:

```
flow 0xa000033c / "main" is max 36;    # then–Teil
flow 0xa00003be / "main" is max 140;   # else–Teil
```

Die Adressen `0xa000033c` und `0xa00003be` repräsentieren Basisblöcke im *then*- bzw. *else*-Pfad der erzeugten *if*-Anweisung.

4.1.3 Durchgeführte Experimente

Loop Nest Splitting wurde für die drei Benchmarks Motion Estimation [GMCG00], QSDPCM [Str88] und Cavity [BTC89] untersucht. Das hier bereits als Beispiel verwendete Motion Estimation und QSDPCM besitzen eine sehr ähnliche Struktur, mit sechs bzw. sieben verschachtelten Schleifen. Cavity enthält nur eine zweifach verschachtelte Schleife, allerdings aufwendigere Berechnungen und mehr *if*-Anweisungen als die anderen beiden Programme.

Für jeden Benchmark wurde der C-Quelltext vor und nach dem Loop Nest Splitting für die beiden untersuchten Architekturen compiliert. Dabei wurde jeweils die höchste Optimierungsstufe der Compiler verwendet (`armcc/tcc -O2` bzw. `tricore-gcc -O3`). Anschließend wurde mit Hilfe von Prozessorsimulatoren eine durchschnittliche Laufzeit gemessen und mit aiT eine WCET-Analyse durchgeführt. Dabei wurden Annotationen zur Beschreibung des Kontrollflusses, wie oben beschrieben, angegeben.

Bei der WCET-Analyse wurde die Länge der *call strings* bei den meisten Experimenten auf 5 begrenzt, lediglich bei Cavity für den ARM7 wurde die Länge auf 3 begrenzt. Es wurden maximal zwei Kontexte pro Schleife unterschieden.

4.1.4 Ergebnisse

WCET und ACET

Abbildung 4.3 zeigt die Auswirkungen des Loop Nest Splitting auf die WCET sowie die ACET der drei Benchmarks, für die drei untersuchten Architekturen. Die WCETs

und ACETs sind relativ zu den Werten der unoptimierten Versionen dargestellt.

Bei der WCET wurden Verbesserungen zwischen 2, 9% (QSDPCM/TriCore) und 89% (ME/Thumb) erreicht. Die Verbesserungen der ACET liegen zwischen 6, 4% (QSDPCM/TriCore) und 74% (ME/Thumb). Für die Benchmarks QSDPCM und Cavity liegen die WCET-Verbesserungen in einer ähnlichen Größenordnung wie die ACET-Verbesserungen. Für den Benchmark ME dagegen ist die Verbesserung der WCET wesentlich höher als die der ACET. Diese Unterschiede zeigen, dass der Einfluss des Loop Nest Splitting auf die WCET stark von der Struktur eines Programmes abhängt.

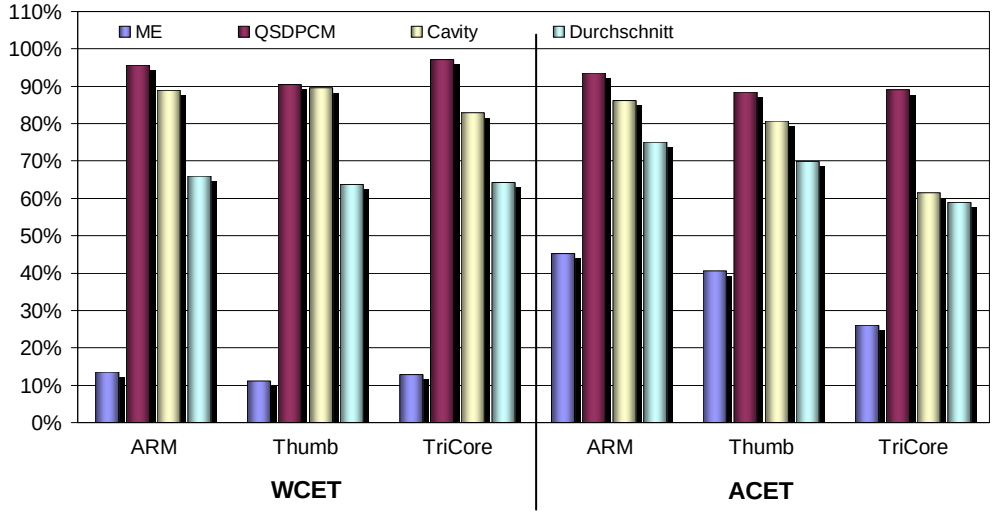

Abbildung 4.3: Relative WCET und ACET nach Loop Nest Splitting

Beim ME Benchmark ist die Überschätzung der WCET vor der Optimierung sehr hoch. Durch die nicht auswertbaren Bedingungen der *if*-Anweisungen in der innersten Schleife liegen für fast alle Iterationen die längeren *else*-Pfade auf dem WC Pfad. Bei einer realen Ausführung des Programms werden allerdings die *then*-Pfade, die nur eine einfache Zuweisung enthalten, sehr viel häufiger ausgeführt als die *else*-Pfade, die einen Arrayzugriff mit einer komplexen Indexberechnung enthalten. Nach dem Loop Nest Splitting wird für einen großen Teil der Iterationen der *then*-Teil der eingefügten *if*-Anweisung ausgeführt, der in der innersten Schleife nur noch die *then*-Teile der beiden *if*-Anweisungen des ursprünglichen Programms enthält. Da durch die angegebenen *flow*-Annotationen die Ausführungsanzahl der *then*- und *else*-Teile der eingefügten *if*-Anweisung genau berücksichtigt wird, enthält auch der WC Pfad viel häufiger die kürzeren *then*-Pfade.

Obwohl QSDPCM eine sehr ähnliche Struktur hat wie ME, sind die WCET-Verbesserungen sehr viel geringer. Dies liegt daran, dass bei QSDPCM, im Gegensatz zu ME, der *then*-Pfad der *if*-Anweisung in der innersten Schleife länger ist als der *else*-Pfad. Dadurch ergibt sich keine so große WCET-Überschätzung, und entspre-

chend geringer sind die Auswirkungen des Loop Nest Splitting.

Die durchschnittliche ACET-Verbesserung über alle Benchmarks beträgt 25% (ARM) bis 41, 1% (TriCore). Die durchschnittliche WCET-Verbesserung liegt zwischen 34% (ARM) und 36, 3% (Thumb), und ist beim ARM und Thumb höher als die entsprechende Verbesserung der ACET.

Einfluss der Annotationen auf die WCET

Abbildung 4.4 verdeutlicht die Notwendigkeit der *flow*-Annotationen für eine exakte WCET-Analyse nach dem Loop Nest Splitting. Im Diagramm ist die WCET der Benchmarks nach Loop Nest Splitting dargestellt, die aiT berechnet, wenn keine zusätzlichen Annotationen, außer den erforderlichen Loop Bounds, angegeben werden. Es sind die prozentualen Änderungen der WCET relativ zu den unoptimierten Versionen dargestellt. Ein Wert von 100% entspricht keiner Veränderung der WCET durch Loop Nest Splitting.

Man sieht, dass die Werte erheblich schlechter sind als die der unoptimierten Versionen. Bei Cavity steigt die WCET auf 113.137% (ARM) bis 135.092% (TriCore) des ursprünglichen Wertes. Diese enormen Verschlechterungen ergeben sich durch die zusätzlich eingefügten Schleifen. Für beide Schleifen mit derselben Schleifenvariablen muss die maximale Anzahl an Iterationen angegeben werden, diese Anzahl multipliziert sich bei der WCET-Analyse, da der Zusammenhang zwischen den Schleifen nicht berücksichtigt werden kann.

Durch die geringere Anzahl an Iterationen der Schleifen sind die Werte bei QSDPCM sehr viel niedriger. Sie liegen aber immer noch zwischen 742, 8% (ARM) und 843, 47% (TriCore).

Beim ME sind die Auswirkungen weniger gravierend, und für den TriCore wird sogar eine Verbesserung der WCET um 60, 7% erreicht. Ein Grund dafür ist, dass die maximale Anzahl an Iterationen der äußeren y-Schleife, durch die Bedingung $\texttt{y} \geq 14$ in der neuen *if*-Anweisung, nur noch 15 beträgt. In der 15-ten Iteration der äußeren y-Schleife ist $y = 14$ und damit die Bedingung $\texttt{x} \geq 10 \;||\; y \geq 14$ der *if*-Anweisung erfüllt. In der inneren y-Schleife wird daraufhin y bis 49 inkrementiert, so dass keine weitere Iteration der äußeren y-Schleife ausgeführt wird (vgl. Abbildung 4.2).

Bei den anderen Benchmarks sind mehrere einzelne Bedingungen durch logische UND Operatoren verknüpft. Mit diesen Bedingungen werden die Schleifenvariablen aller umgebenden Schleifen geprüft. Es existieren also Iterationen der äußersten Schleife, in denen die neue *if*-Anweisung nie erfüllt ist. Für diese Iterationen hat die Schleife, die die *if*-Anweisung enthält, weiterhin die maximale Iterationsanzahl wie vor dem Loop Nest Splitting, da keine Iterationen durch die zusätzliche Schleife im *then*-Pfad der *if*-Anweisung übernommen werden, wie dies bei ME immer der Fall ist.

Der zweite Grund für die Verbesserung beim TriCore ist, dass der `tricore-gcc`

den *then*-Pfad erheblich vereinfacht. Die innerste Schleife entfällt dabei komplett. Dadurch ist der *then*-Pfad, trotz zu vieler gezählter Iterationen, kürzer als der *else*-Pfad.

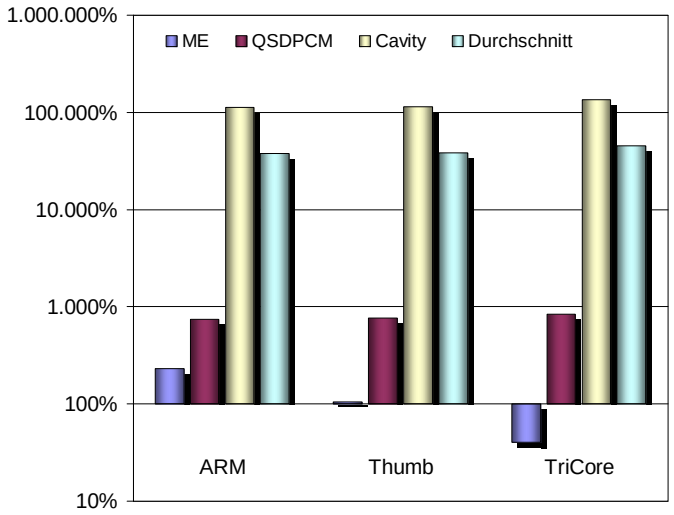

Abbildung 4.4: Relative WCET nach Loop Nest Splitting ohne flow Annotationen

Codegröße

Natürlich erhöht sich durch das Loop Nest Splitting auch die Codegröße der Programme. Die relativen Codegrößen nach der Optimierung sind in Abbildung 4.5 dargestellt. Das Diagramm zeigt die prozentuale Codegröße nach Loop Nest Splitting. Die Codegröße vor der Optimierung entspricht 100%. Diese Werte wurden berechnet auf Basis der Größe der Text-Segmente der ausführbaren ELF-Dateien der Benchmarks.

Die Codegröße steigt bei Cavity um $11,2\%$ (TriCore) bis $19,9\%$ (ARM). Diese Werte sind sehr hoch, da die Schleifen im Gegensatz zu den anderen Benchmark sehr viel Code enthalten. Bei QSDPCM steigt die Codegröße um $4,11\%$ (TriCore) bis $10,5\%$ (Thumb). Bei ME sind die Zuwächse am geringsten, obwohl die größte Verbesserung bei der WCET erreicht wurde. Sie liegen zwischen $2,3\%$ (TriCore) und $8,1\%$ (Thumb).

Für eingebettete Systeme spielt neben der WCET häufig auch die Codegröße ein wichtige Rolle. Mit Loop Nest Splitting ist auch eine Abwägung zwischen Reduzierung der WCET/ACET und der Codegröße möglich. Normalerweise wird die *if*-Anweisung zur Aufteilung des Iterationsraumes in die innerste Schleife eingefügt, von der die Bedingungen abhängig sind. Für das Beispiel aus Abbildung 4.2 ist das die y Schleife, da die Bedingung von x und y abhängt. Da die *if*-Anweisung nicht von den Schleifenvariablen der weiter innen liegenden Schleifen abhängt, kann sie aber

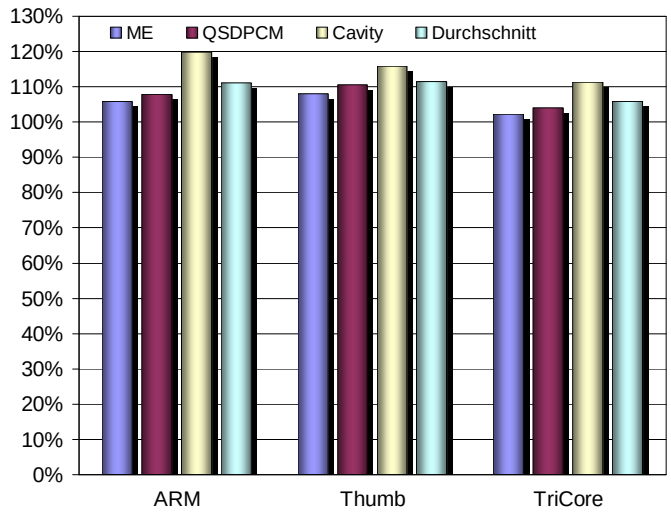

Abbildung 4.5: Relative Codegröße nach Loop Nest Splitting

auch in eine dieser Schleifen eingefügt werden. Dadurch verringert sich die Codegröße, da weniger Code dupliziert wird. Allerdings steigen die WCET und die ACET, da die *if*-Anweisung häufiger ausgeführt wird.

In Abbildung 4.6 ist der Zusammenhang zwischen Codegröße und WCET für die Benchmarks ME und QSDPCM dargestellt. Jede Datenreihe besteht aus fünf Punkten, die jeweils die Aufteilung des Iterationsraumes in einer der fünf möglichen Schleifen repräsentieren. Die x-Achse stellt die prozentuale Änderung der WCET dar, die y-Achse die prozentuale Änderung der Codegröße. Der Wert 100% entspricht der WCET bzw. Codegröße der unoptimierten Version.

Bei ME sind die Änderungen der WCET relativ gering. Für den ARM-Instruktionssatz des ARM7 liegt die WCET der fünf unterschiedlich aufgeteilten Versionen zwischen $13,5\%$ und $16,2\%$ der WCET der unoptimierten Version. Die Codegröße wird dabei um $1,5\%$ bis $5,8\%$ erhöht. Bei allen drei Architekturen steigt die WCET, je weiter innen die *if*-Anweisung eingefügt wird. Gleichzeitig sinkt die Erhöhung der Codegröße.

Bei QSDPCM sind die Änderungen der WCET deutlicher. Bei einer Platzierung der *if*-Anweisung in der inneren Schleifen kommt es sogar zu einer Verschlechterung der WCET. Für den ARM-Instruktionssatz des ARM7 liegt die WCET nach dem Loop Nest Splitting zwischen $95,6\%$ und $120,1\%$ der WCET der unoptimierten Version. Die Erhöhung der Codegröße liegt zwischen $3,2\%$ und $7,9\%$. Beim Thumb-Instruktionssatz des ARM7 und beim TriCore erhöht sich die WCET nicht in allen Fällen bei einer Platzierung der *if*-Anweisung in einer weiter innen liegenden Schleife. Diese Ausnahmen werden verursachte durch unterschiedliche Codeerzeugung und Registerallokation durch die Compiler, bei der verschiedenen Versionen.

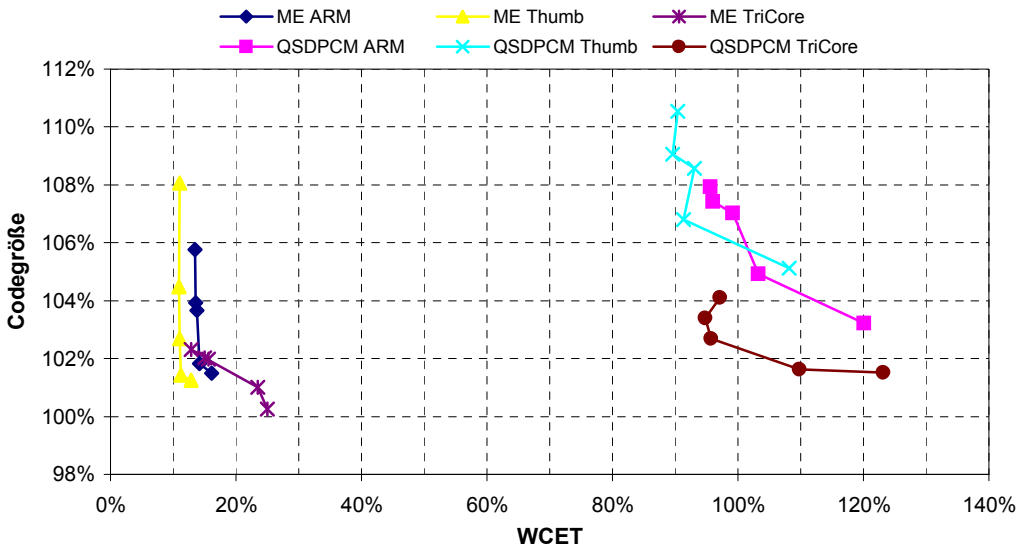

Abbildung 4.6: Abwägung zwischen WCET und Codegröße beim Loop Nest Splitting

Die hier vorgestellten Ergebnisse der Untersuchung der WCET-Optimierung durch Loop Nest Splitting wurden auch in den zwei Arbeiten [FS06a] und [FS06b] separat veröffentlicht.

4.2 Procedure Cloning

Als zweite Kontrollfluss-Optimierung wurde das Procedure Cloning [BGS94] untersucht. Bei dieser Optimierung werden Kopien von Funktionen erzeugt, wobei ein oder mehrere Parameter der Funktion durch Konstanten ersetzt werden. So entstehen verschiedene, spezialisierte Versionen einer Funktion. Aufrufe der ursprünglichen Funktion können dann durch Aufrufe einer passenden spezialisierten Version ersetzt werden.

Durch das Klonen von Funktionen kann auf verschiedene Weisen eine Verbesserung der Laufzeit erreicht werden. Da Variablen (Funktionsparameter) durch Konstanten ersetzt werden, können in den geklonten Funktionen häufig durch Constant Propagation, Constant Folding und Copy Propagation weitere Instruktionen eingespart werden. Möglicherweise können sogar ganze Kontrollstrukturen entfallen, wie z.B. *if*-Anweisungen, deren Bedingung nur noch aus Konstanten besteht, oder Schleifen mit konstanter Abbruchbedingung und genau einer Iteration. Außerdem kann das Procedure Cloning weitere Optimierungen in den verschiedenen Kopien ermöglichen, die in der allgemeinen Version noch nicht möglich waren, wie z.B. das Ersetzen von Berechnungen durch weniger aufwendigere (siehe Abbildung 4.7).

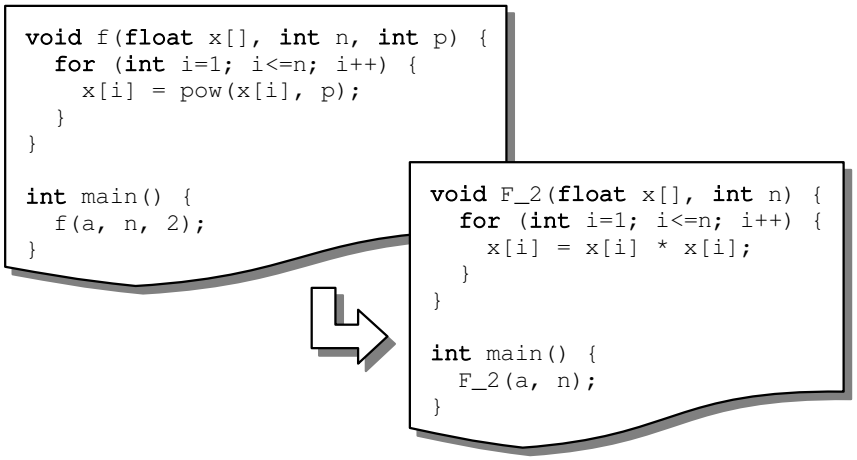

Abbildung 4.7: Beispiel Procedure Cloning

Zusätzlich zu den Einsparungen innerhalb der geklonten Funktionen wird der Aufwand der Funktionsaufrufe reduziert, da weniger Parameter übergeben werden.

4.2.1 Auswirkungen auf die WCET

Für die WCET-Analyse sind noch weitere positive Auswirkungen durch das Procedure Cloning möglich. Falls in einer spezialisierten Funktion eine *if*-Anweisung entfallen kann, verbessert sich die WCET, wie auch die ACET, durch das Wegfallen der Instruktionen zur Auswertung der Bedingung sowie der Sprünge. Zusätzlich kann der vereinfachte Kontrollfluss zu einer genaueren WCET-Abschätzung führen.

Das in Abbildung 4.8 dargestellte Beispiel enthält in der Funktion `f1` eine *if*-Anweisung, deren Bedingung vom Parameter `x` abhängt. Falls die Bedingung nicht durch die Value-Analyse ausgewertet werden kann, beispielsweise da der auf dem Stack übergebene Wert durch einen schreibenden Speicherzugriff auf eine unbekannte Adresse verloren ging, wird der `then_block`, auch für den Aufruf mit $x = 5$, auf dem WC Pfad liegen. Nach dem Procedure Cloning kann die gesamte *if*-Anweisung entfernt werden, und dieser Fehler bei der WCET-Abschätzung tritt nicht mehr auf.

Ein weiterer Vorteil der geklonten Funktionen für die WCET-Analyse ergibt sich aus der Möglichkeit, eigene Annotationen für die verschiedenen Kopien der Funktion anzugeben. Benutzerangaben, die bei aiT in einer sog. ais-Datei oder im C-Quelltext angegeben werden können, gelten immer für alle Kontexte. Enthält eine Funktion beispielsweise eine Schleife, deren Loop Bounds von einem Funktionsparameter abhängen, so müsste in der Loop Bound Annotation der maximal mögliche Wert angegeben werden, der dann allerdings für alle Aufrufe verwendet wird. Die von aiT durchgeführte Loop Bound Analyse ermittelt hingegen die Loop Bounds von Schleifen für jeden betrachteten Kontext separat. Können die Loop Bounds jedoch nicht

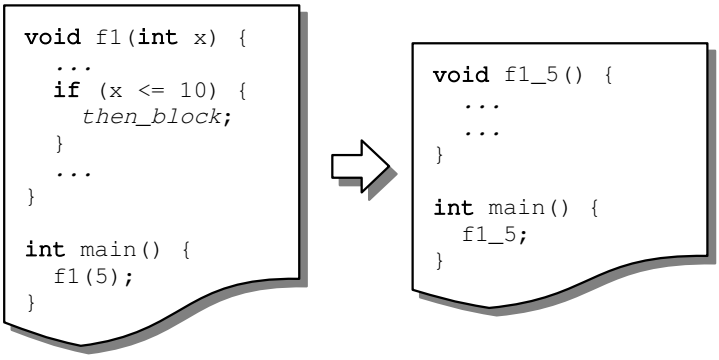

Abbildung 4.8: Beispiel Procedure Cloning (if-Anweisung)

für alle Kontexte automatisch ermittelt werden, überschreiben die dann nötigen Benutzerangaben auch die automatisch ermittelten Werte für einige Kontexte.

Das Beispiel in Abbildung 4.9 verdeutlicht die Auswirkungen auf die WCET. Die Funktion `f2` wird dreimal aufgerufen, zweimal mit dem Wert 5 und einmal mit dem Wert 20 für den Parameter `x`. Die Schleife in der Funktion hat also bei den ersten beiden Aufrufen 25 Iterationen und beim dritten Aufruf 100. Können die Loop Bounds nicht automatisch ermittelt werden, müssten diese mit maximal 100 Iterationen angegeben werden. Dies hat zur Folge, dass auch für die ersten beiden Aufrufe 100 Iterationen bei der Bestimmung des WC Pfades gezählt werden.

Im Beispiel wurde eine spezialisierte Version der Funktion `f2` mit dem Wert 5 für den Parameter `x` erstellt. Die beiden ersten Aufrufe der Funktion `f2` wurden durch Aufrufe der neuen Funktion `f2_5` ersetzt. Der dritte Aufruf ruft weiterhin die ursprüngliche Funktion auf. Die *for*-Schleife in der Funktion `f2_5` hat immer genau 25 Iterationen. Dies kann nun durch Loop Bound Annotationen für diese Funktion angegeben werden. Damit werden bei den ersten beiden Aufrufen nun 25 Iterationen berücksichtigt und lediglich beim dritten Aufruf 100 Iterationen, wodurch die WCET-Überschätzung erheblich reduziert wurde.

4.2.2 Auswahl der Funktionen

Bei der Auswahl von Kandidaten für das Procedure Cloning müssen mehrere Faktoren berücksichtigt werden. Durch das Kopieren der Funktionen erhöht sich auch die Codegröße. Die Funktionen sollten also nicht zu groß sein, bzw. die Auswirkungen auf die Codegröße müssen gegen eine mögliche Verbesserung der Laufzeit abgewogen werden.

Desweiteren muss das Programm Aufrufe der zu klonenden Funktion mit Konstanten als Parametern enthalten. Es wäre auch denkbar, Informationen der Value-Analyse zu verwenden. Diese berechnet die Registerwerte zu jedem Programmpunkt, also auch die Werte der Parameter bei Funktionsaufrufen. Könnte beim Procedu-

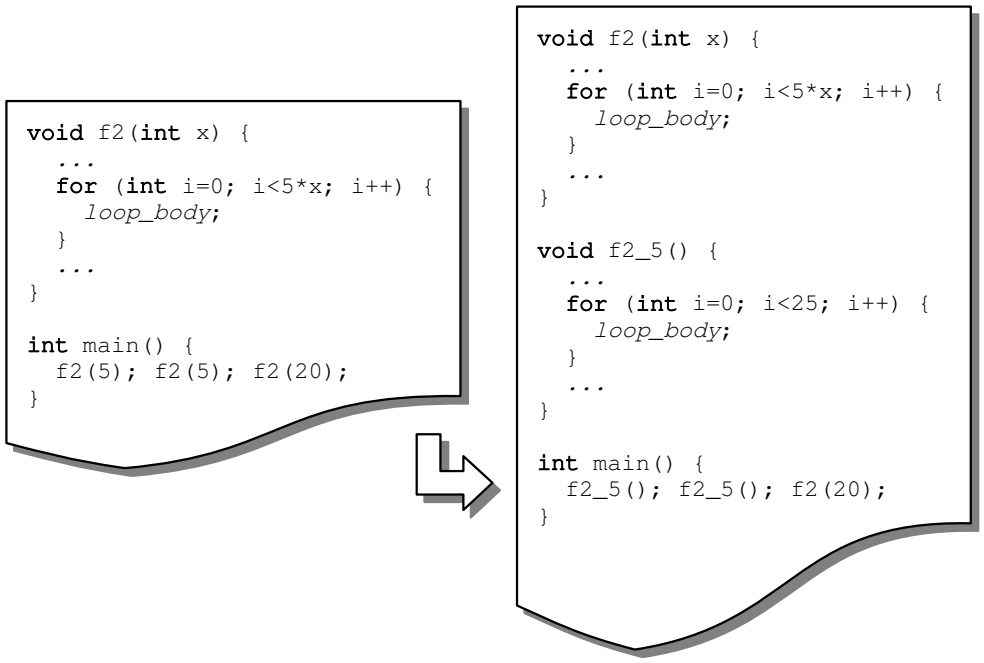

Abbildung 4.9: Beispiel Procedure Cloning (for-Schleife)

re Cloning auf diese Werte zurückgegriffen werden, könnten auch Funktionsaufrufe, die eine Variable als Parameter enthalten, deren Wert immer gleich ist und durch die Value-Analyse berechnet wird, durch Aufrufe spezialisierter Funktionen ersetzt werden. Leider scheint es bei aiT aber keine Möglichkeit zu geben, an diese Informationen zu gelangen, da diese nur intern verwendet werden.

Um eine möglichst große Reduzierung der Laufzeit zu erreichen, sollten möglichst häufig ausgeführte Funktionen gewählt werden, die einen großen Einfluss auf die gesamte Laufzeit des Programms haben. Bei der Reduzierung der durchschnittlichen Laufzeit werden dazu üblicherweise Profiling Informationen verwendet. Analog können bei der WCET-Optimierung Informationen über die Ausführungsanzahl und WCET der Funktionen verwendet werden, die durch eine WCET-Analyse des Programms vor dem Procedure Cloning gewonnen werden.

4.2.3 Annotationen für die WCET-Analyse

Nach der Durchführung des Procedure Cloning müssen auch für die erzeugten Kopien einer Funktion die nötigen Annotationen angegeben werden. Diese entsprechen im Wesentlichen denen der ursprünglichen Funktion, allerdings können z.B. Loop Bounds oft für die spezialisierten Funktionen genauer angegeben werden, als für die allgemeine Funktion, wie im bereits vorgestellten Beispiel aus Abbildung 4.9.

Auch kann es vorkommen, dass durch das Ersetzen eines Parameters durch eine

Konstante die Loop Bound Analyse nach der Optimierung in der Lage ist, die Loop Bounds einer Schleife zu ermitteln, so dass keine Benutzerangaben mehr notwendig sind.

4.2.4 Durchgeführte Experimente

Zur Untersuchung des Procedure Cloning wurden die drei Benchmarks EPIC, MPEG2 und GSM aus der MediaBench Suite [LPMS97] betrachtet. Sowohl die Auswahl der Funktionen als auch die Durchführung der Optimierung auf dem C-Quelltext der Benchmarks, inklusive Constant Folding und Constant Propagation, wurden manuell durchgeführt. Gleiches gilt für das Ermitteln der Loop Bounds vor und nach der Optimierung. Die Auswahl der Funktionen wurde hier allerdings nicht anhand von WCET-Informationen durchgeführt.

Bei dem Benchmark EPIC wurden die beiden Funktionen `internal_filter` und `reflect1` geklont. Es wurden jeweils zwei Versionen erzeugt, und die originalen Funktionen wurden nicht mehr benötigt. Die Funktion `internal_filter` ist sehr groß. Sie enthält insgesamt 32 Schleifen, die bis zu vierfach verschachtelt sind. Die Loop Bounds sind abhängig von den Parametern der Funktion und können nicht automatisch von aiT ermittelt werden. Das folgende Code-Fragment zeigt die Parameter der originalen Funktion `internal_filter`:

```
internal_filter(float *image, int x_dim, int y_dim,
                float *filt, float *temp, int x_fdim,
                int y_fdim, int xgrid_start, int xgrid_step,
                int ygrid_start,int ygrid_step, float *result) {
  ...
}
```

Das Programm enthält sechs aufeinanderfolgende Aufrufe dieser Funktion mit unterschiedlichen (konstanten) Werten für einige der Parameter:

```
internal_filter (image, level_x_size, level_y_size,
                 lo_filter, filtertemp, filter_size, 1, 0,
                 2, 0, 1, lo_imagetemp);
internal_filter (image, level_x_size, level_y_size,
                 hi_filter, filtertemp, filter_size, 1, 1,
                 2, 0, 1, hi_imagetemp);
internal_filter (lo_imagetemp, level_x_size, level_y_size,
                 lo_filter, filtertemp, 1, filter_size, 0,
                 1, 0, 2, result_block);
internal_filter (lo_imagetemp, level_x_size, level_y_size,
                 hi_filter, filtertemp, 1, filter_size, 0,
                 1, 1, 2, (result_block += (total_size/4)));
internal_filter (hi_imagetemp, level_x_size, level_y_size,
                 lo_filter, filtertemp, 1, filter_size, 0,
                 1, 0, 2, (result_block += (total_size/4)));
internal_filter (hi_imagetemp, level_x_size, level_y_size,
```

```
                  hi_filter, filtertemp, 1, filter_size, 0,
                  1, 1, 2, (result_block += (total_size/4)));
```

In den spezialisierten Versionen `internal_filter1` und `internal_filter2` wurden jeweils vier der Parameter durch Konstanten ersetzt. Das folgende Code-Fragment zeigt die Parameterliste dieser beiden Funktionen:

```
internal_filter1(float *image, int x_dim, int y_dim,
                 float *filt, float *temp, int x_fdim,
                 int xgrid_start, float *result) {
  ...
}

internal_filter2(float *image, int x_dim, int y_dim,
                 float *filt, float *temp, int y_fdim,
                 int ygrid_start, float *result) {
  ...
}
```

Die sechs Aufrufe der Funktion `internal_filter` wurden entsprechend durch Aufrufe der neuen Funktionen ersetzt:

```
internal_filter1 (image, level_x_size, level_y_size,
                  lo_filter, filtertemp, filter_size,
                  0, lo_imagetemp);
internal_filter1 (image, level_x_size, level_y_size,
                  hi_filter, filtertemp, filter_size,
                  1, hi_imagetemp);

internal_filter2 (lo_imagetemp, level_x_size, level_y_size,
                  lo_filter, filtertemp, filter_size,
                  0, result_block);
internal_filter2 (lo_imagetemp, level_x_size, level_y_size,
                  hi_filter, filtertemp, filter_size,
                  1, (result_block += (total_size/4)));
internal_filter2 (hi_imagetemp, level_x_size, level_y_size,
                  lo_filter, filtertemp, filter_size,
                  0, (result_block += (total_size/4)));
internal_filter2 (hi_imagetemp, level_x_size, level_y_size,
                  hi_filter, filtertemp, filter_size,
                  1, (result_block += (total_size/4)));
```

Die zweite geklonte Funktion `reflect1` wird mehrmals innerhalb der Funktion `internal_filter` aufgerufen. Die Funktion wird sowohl bei der Komprimierung als auch bei der Dekomprimierung verwendet. Die gewünschte Aktion wird durch einen Parameter (`f_or_e`) angegeben, und bestimmte Teile der Funktion werden in Abhängigkeit von diesem Parameter ausgeführt:

```
reflect1(float *filt, int x_dim, int y_dim, int x_pos,
         int y_pos, float *result, int f_or_e)
```

```
{
  ...
  if (f_or_e IS EXPAND) {
    ...
  }

  ...

  if (f_or_e IS EXPAND) {
    ...
  }
}
```

Für diese Funktion wurden zwei Versionen erstellt, eine für die Komprimierung und eine für die Dekomprimierung. So konnten die *if*-Anweisungen zur Unterscheidung der beiden Fälle entfallen.

Bei dem zweiten Benchmark MPEG2 wurden ebenfalls zwei Funktionen geklont. Die erste Funktion `fullsearch` besitzt einen Parameter, der die Höhe eines Blockes in Pixeln angibt. Bei den verschiedenen Aufrufen der Funktion werden die Werte 8 und 16 für die Blockhöhe verwendet. Deshalb wurden von dieser Funktion zwei Versionen für diese beiden Werte erstellt. Innerhalb der Funktion `fullsearch` wird mehrmals die Funktion `dist1` aufgerufen, der ebenfalls die Blockgröße übergeben wird. Nach dem Klonen sind diese Werte Konstanten, nämlich 8 oder 16, somit können auch für die Funktion `dist1` zwei Versionen für die beiden Blockgrößen erstellt werden. Diese Funktion enthält mehrere Schleifen, deren Anzahl an Iterationen der jeweiligen Blockgröße entspricht.

Beim Benchmark GSM wurde die Funktion `Short_term_analysis_filtering` geklont. Die WCET dieser Funktion macht einen großen Teil der gesamten WCET aus. Sie enthält eine Schleife, deren Iterationsanzahl direkt abhängt von einem der Funktionsparameter. Die Funktion wird viermal aufgerufen, mit den Werten 13, 14, 13 und 120 für diesen Parameter. Die Loop Bounds müssen entsprechend mit maximal 120 Iterationen angegeben werden. Es wurde eine Kopie dieser Funktion erstellt, die von den ersten drei Aufrufen verwendet wird. Dabei wurde kein Parameter durch eine Konstante ersetzt, da die neue Funktion noch mit den unterschiedlichen Werten 13 und 14 aufgerufen wird. Für die Kopie der Funktion können nun aber die Loop Bounds mit maximal 14 Iterationen angegeben werden.

Wie bereits beim Loop Nest Splitting wurden die Benchmarks vor und nach der Optimierung mit der höchsten Optimierungsstufe kompiliert, um anschließend mit aiT eine WCET-Abschätzung zu berechnen und eine Ausführungszeit mit dem Prozessorsimulator zu ermitteln. Allerdings wurden die Experimente nur für die ARM-Architektur durchgeführt, da die Benchmarks sehr umfangreich sind und die WCET-Analyse für den komplexeren TriCore Prozessor nicht in vertretbarer Zeit bzw. mit dem vorhandenen Speicherplatz möglich war.

Bei der WCET-Analyse wurde die Länge der *call strings* für die Benchmarks EPIC und MPEG2 auf 3 begrenzt, und für GSM auf 7. Die maximale Anzahl an Kontexten pro Schleife wurde für EPIC und MPEG2 auf 1 gesetzt, und für GSM auf 3.

4.2.5 Ergebnisse

WCET und ACET

In Abbildung 4.10 ist die prozentuale Verbesserung der WCET sowie der ACET nach dem Procedure Cloning dargestellt. Ein Wert von 100% entspricht der WCET bzw. ACET der unoptimierten Version. Bei EPIC wurde die ACET um $1,8\%$ (ARM) und 3% (Thumb) reduziert, bei MPEG2 um $1,3\%$ (ARM) und $2,6\%$ (Thumb). Bei GSM gab es keine nennenswerte Änderung der ACET. Die durchschnittliche Reduzierung der ACET über alle Benchmarks beträgt 1% (ARM) bzw. $1,8\%$ (Thumb).

Die Auswirkungen auf die WCET sind hingegen sehr viel größer. Die mit Abstand größte Verbesserung wurde bei EPIC erreicht, mit einer Reduzierung der WCET um $95,7\%$ (ARM) und $95,6\%$ (Thumb). Auch bei MPEG2 wurden mit $33,2\%$ (ARM) und $33,4\%$ (Thumb) erhebliche Verbesserungen erreicht. Bei GSM gab es beim ARM Instruktionssatz keine Änderung der WCET, aber beim Thumb Instruktionssatz eine Reduzierung um 31%. Die durchschnittliche WCET-Verbesserung beträgt 43% (ARM) bzw. $53,3\%$ (Thumb).

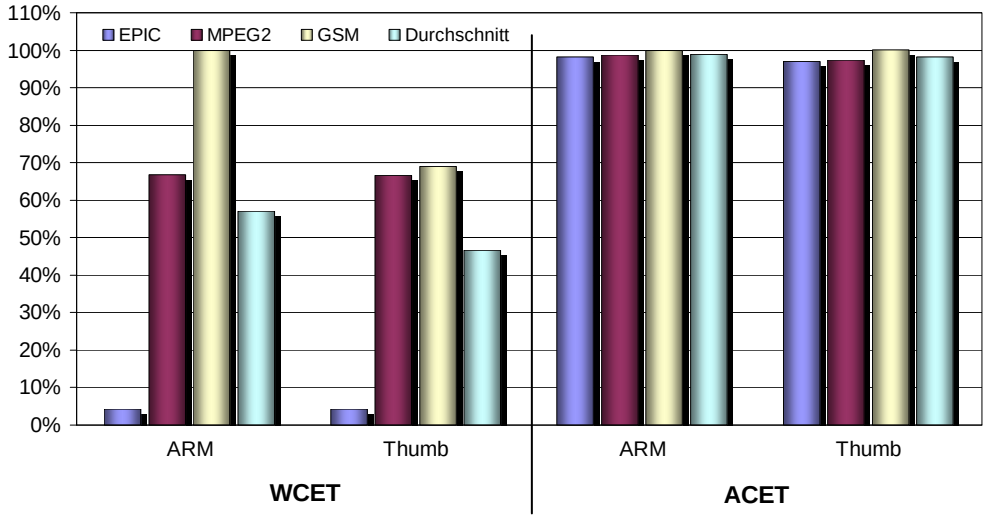

Abbildung 4.10: Relative WCET und ACET nach Procedure Cloning

Die enorm hohe Reduzierung der WCET bei EPIC ist im Wesentlichen auf die genauere Angabe der Loop Bounds für die Funktion `internal_filter` nach dem Procedure Cloning zurückzuführen. Für einige der Schleifen variiert die Anzahl der Iterationen sehr stark bei den verschiedenen Aufrufen. Dadurch werden bei der Be-

stimmung des WC Pfades viel zuviele Iterationen gezählt und es entsteht eine hohe WCET-Überschätzung. Dieser Effekt wird noch verstärkt durch viele Gleitkommaberechnungen in den Schleifen. Da der ARM7TDMI keine FPU besitzt, werden für Gleitkommaberechnungen Bibliotheksfunktionen verwendet. Diese haben meist eine komplizierte Struktur mit vielen Verzweigungen, wodurch sich ein großer Fehler bei der WCET-Abschätzung ergibt. Nach dem Klonen der Funktion `internal_filter` können die Loop Bounds genauer angegeben werden, und der Fehler durch zuviel gezählte Iterationen wird stark verringert.

Auch bei MPEG2 kommt die WCET-Reduzierung hauptsächlich durch die genauere Angabe der Loop Bounds für eine Kopie der Funktion `dist1` zustande. Dies hat so große Auswirkungen auf die WCET des Programms, da die WCET dieser Funktion über 99% der gesamten WCET ausmacht. Dieser hohe Anteil ergibt sich dadurch, dass die Funktion aus einer verschachtelten Schleife heraus aufgerufen wird, für die nur sehr ungenaue Loop Bounds angegeben werden konnten. Dadurch werden bei der Pfad Analyse über 1,4 Milliarden Aufrufe dieser Funktion gezählt.

Die sehr unterschiedlichen Ergebnisse für die ARM und den Thumb Instruktionssätze bei dem Benchmark GSM kommen dadurch zustande, dass die Loop Bound Analyse von aiT bei der für dem ARM Instruktionssatz compilierten Version die Loop Bounds für die Schleife in der geklonten Funktion erkennt. D.h. es wird für jeden Aufruf der Funktion die genaue Anzahl an Iterationen berücksichtigt, und das Klonen der Funktion bringt keine Verbesserung. Beim Thumb Instruktionssatz werden die Loop Bounds nicht automatisch erkannt, und nach dem Procedure Cloning ist, wie bei den anderen Benchmarks, eine genauere Angabe der Loop Bounds möglich als in der ursprünglichen Version.

Da aiT intern kontextabhängige Loop Bounds unterstützt, wäre das Klonen von Funktionen eigentlich nicht nötig, um exaktere Loop Bounds anzugeben. Allerdings gelten die üblichen Benutzerangaben immer für alle Kontexte. Würden die Loop Bound Informationen direkt in die CRL Zwischendarstellung von aiT eingefügt, könnten vermutlich ähnliche Verbesserungen ohne das Duplizieren von Code erreicht werden.

Überschätzung der WCET

Die hier beschriebenen Verbesserungen der WCET basieren zum größten Teil auf einer Reduzierung der Überschätzung durch genauere Angabe der Loop Bounds. Obwohl die Überschätzung verringert wird, ist sie, insbesondere bei EPIC und MPEG2, auch nach der Optimierung noch sehr hoch. In Abbildung 4.11 ist das Verhältnis der WCET zur ACET vor und nach dem Procedure Cloning dargestellt. Bei EPIC beträgt die WCET vor der Optimierung das 165-fache der ACET (ARM), nach der Optimierung noch das 7-fache. Bei MPEG2 dagegen wird das Verhältnis nur von vorher 26.082 auf 17.634 reduziert. Obwohl sich von der gemessenen Laufzeit einer Ausführung nicht auf die theoretische Worst-Case Laufzeit schließen lässt, ist auch

nach der Optimierung noch von einer hohen Überschätzung der WCET auszugehen.

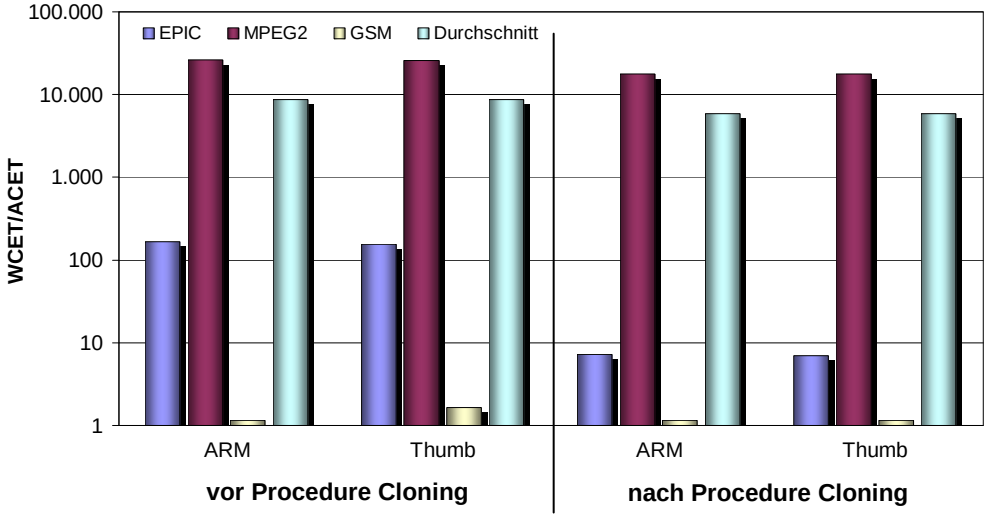

Abbildung 4.11: WCET relativ zur ACET vor und nach Procedure Cloning

Die enorme Überschätzung bei MPEG2 entsteht im Wesentlichen durch eine zweifach verschachtelte Schleife in der häufig aufgerufenen Funktion `fullsearch`. Die Grenzen des Iterationsbereiches der Schleifen werden zuvor in einer weiteren Schleife, abhängig von Funktionsparametern und Eingabedaten, berechnet. Für die Loop Bounds müssen deshalb sehr hohe Maximalwerte angegeben werden.

Bei GSM sind die Unterschiede zwischen WCET und ACET sehr viel geringer. Beim ARM-Instruktionssatz beträgt das Verhältnis der WCET zur ACET vor sowie nach der Optimierung $1,2$. Beim Thumb-Instruktionssatz wird das Verhältnis von $1,7$ vor der Optimierung auf $1,1$ nach der Optimierung verbessert. Die WCET-Überschätzung ist bei GSM also nur sehr gering.

Codegröße

In Abbildung 4.12 sind die Auswirkungen des Procedure Cloning auf die Codegröße dargestellt. Dargestellt ist die prozentuale Codegröße nach der Optimierung, wobei der Wert 100% der Codegröße der unoptimierten Version entspricht. Da bei GSM nur eine relativ kleine Funktion dupliziert wurde, stiegt die Codegröße nur um $0,7\%$ (ARM, Thumb). Bei MPEG2 wurden zwei größere Funktionen geklont, was zu einer Erhöhung der Codegröße um $12,4\%$ (ARM) bzw. $9,7\%$ (Thumb) führte. Die Zuwächse bei EPIC betragen $8,1\%$ (ARM) und $6,4\%$ (Thumb).

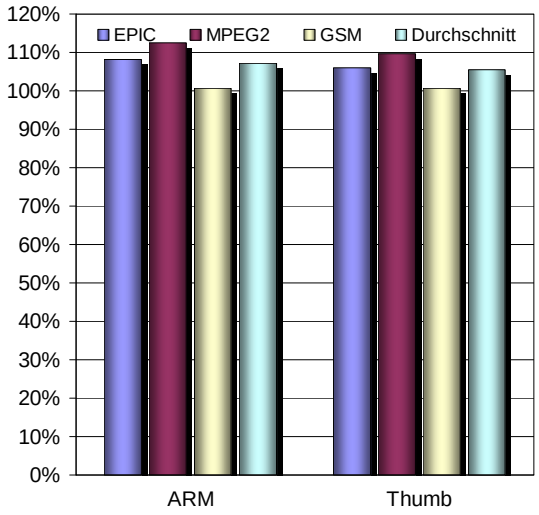

Abbildung 4.12: Relative Codegröße nach Procedure Cloning

4.3 Loop Unrolling

Die dritte hier betrachtete Kontrollfluss-Optimierung ist das Loop Unrolling. Beim Loop Unrolling werden Schleifen ganz oder teilweise „abgerollt“, d.h. es werden u Kopien des Schleifenkörpers erzeugt, und die Anzahl der Iterationen der Schleife wird angepasst, indem die Schleifenvariable bei jeder Iteration um u inkrementiert wird, anstatt um 1. Der Wert u wird als Unrolling-Faktor bezeichnet. In Abbildung 4.13 ist ein Beispiel für das zweifache Abrollen einer Schleife dargestellt.

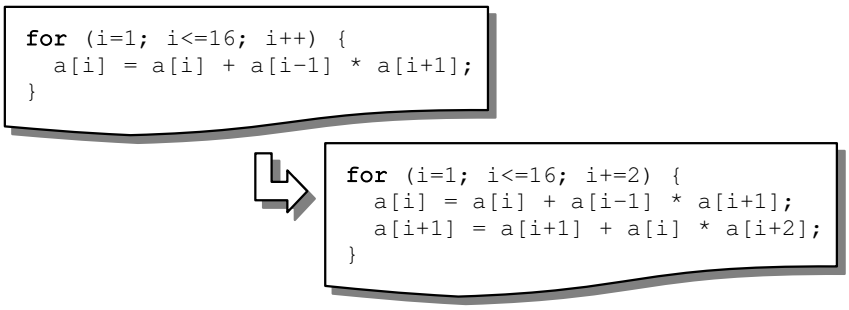

Abbildung 4.13: Beispiel Loop Unrolling (konstante Iterationsanzahl)

Durch das Abrollen von Schleifen kann der Schleifenoverhead reduziert werden. Da weniger Iterationen ausgeführt werden, werden auch weniger Sprünge durchgeführt und somit *pipeline stalls* durch Kontrollhazards vermieden. Außerdem kann sich die Parallelisierbarkeit der Instruktionen in der Schleife erhöhen, wodurch die Auslastung der Ausführungseinheiten des Prozessors erhöht wird. Desweiteren können in bestimmten Fällen Speicherzugriffe vermieden werden, wenn beispielsweise

Arrayelemente innerhalb einer Iteration mehrfach benutzt werden können, wie im Beispiel aus Abbildung 4.13.

Dieses einfache Vorgehen ist jedoch nur möglich, wenn die Anzahl der Iterationen durch den Unrolling-Faktor teilbar ist. Für allgemeine Schleifen mit einer variablen Iterationsanzahl n können nur die ersten $\lfloor n/u \rfloor$ Iterationen durch die abgerollte Schleife ausgeführt werden. Zur Behandlung der restlichen (maximal $u-1$) Iterationen ist eine zusätzliche Schleife erforderlich (bzw. eine *if*-Anweisung für $u = 2$). Abbildung 4.14 zeigt ein Beispiel für das Abrollen einer *for*-Schleife mit n Iterationen und $u = 4$.

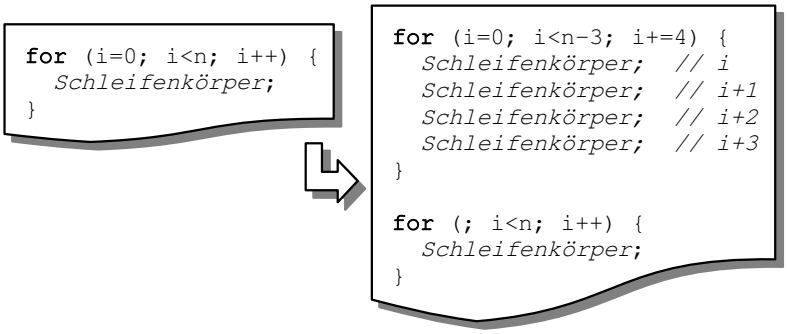

Abbildung 4.14: Beispiel Loop Unrolling (variable Iterationsanzahl)

Durch die zusätzliche Schleife entsteht allerdings auch zusätzlicher Overhead. Für Schleifen mit einer geringen Anzahl an Iterationen kann dieser u.U. größer sein als die durch das Abrollen erreichte Verbesserung.

Bei der Betrachtung der Auswirkungen des Loop Unrolling auf die WCET muss unterschieden werden zwischen dem Fall einer konstanten Iterationsanzahl und dem einer variablen bzw. unbekannten Iterationsanzahl. Ist die Iterationsanzahl konstant, so dürften die Auswirken auf die WCET i.W. identisch sein mit denen auf die durchschnittliche Laufzeit. Bei einer variablen Iterationsanzahl muss jedoch eine zusätzliche Schleife eingefügt werden, die häufig negative Auswirkungen auf die WCET-Analyse hat.

4.3.1 Auswahl zu optimierender Schleifen

Die Auswahl der Schleifen für das Loop Unrolling sowie der zugehörigen Unrolling-Faktoren muss unter Berücksichtigung der aus der Optimierung resultierenden Erhöhung der Codegröße erfolgen. Um möglichst große Auswirkungen auf die Gesamtlaufzeit zu erreichen, sollten häufig ausgeführte Schleifen gewählt werden, bzw. im Kontext der WCET-Optimierung Schleifen mit einer hohen Ausführungsanzahl in dem bei der WCET-Analyse ermittelten Worst-Case Szenario. Diese Werte können sich durchaus stark unterscheiden, so dass das Abrollen einer Schleife, die üblicher-

weise nur selten ausgeführt wird, auf dem WC Pfad aber sehr viel häufiger vorkommt, stärkere Auswirkungen auf die WCET hat als auf die ACET.

Bei Schleifen mit variabler Iterationsanzahl sollte die Anzahl der Iterationen nicht zu gering sein, damit der Overhead durch die zusätzliche Schleife nicht zu stark ins Gewicht fällt. Für die WCET-Optimierung können dabei die bei der WCET-Analyse berücksichtigten Loop Bounds betrachtet werden. Dies ist ein großer Vorteil gegenüber der ACET-Reduzierung, da diese Werte immer genau bekannt sind.

4.3.2 Annotationen für die WCET-Analyse

Nach dem Abrollen einer Schleife müssen die Loop Bound Annotationen für diese Schleife entsprechend angepasst werden. Dazu muss lediglich die maximale Iterationsanzahl der ursprünglichen Schleife durch den verwendeten Unrolling-Faktor dividiert werden.

Wurde eine zusätzliche Schleife zur Behandlung der letzten Iterationen eingefügt, so muss auch für diese eine Loop Bound Annotation angegeben werden. Die Loop Bounds für diese Schleife werden i.d.R. nicht automatisch erkannt, da sie abhängig sind vom Wert der Schleifenvariablen nach der Ausführung der abgerollten Schleife. Wenn nicht alle Iterationen der Schleife in einem eigenen Kontext betrachtet werden, kann die Value Analyse diesen Wert nicht bestimmen.

Für eine Schleife mit n Iterationen, die um den Faktor u abgerollt wurde, beträgt die Anzahl der Iterationen der zusätzlichen Schleife ($n \bmod u$), d.h. maximal $u - 1$. Dieser Maximalwert muss also in Loop Bound Annotationen für die Schleife angegeben werden. Dadurch wird aber diese maximale Anzahl an Iterationen für jeden Aufruf der Schleife gezählt und es entsteht eine hohe WCET-Überschätzung. Je höher der Unrolling-Faktor ist, desto gravierender sind die Auswirkungen dieser Überschätzung.

Wird beispielsweise eine Schleife mit maximal 41 Iterationen um den Faktor 8 abgerollt, so beträgt die maximale Iterationsanzahl der abgerollten Schleife 5, und die der zusätzlichen Schleife 7. Bei jeder Ausführung der Schleifen würden also $5 \cdot 8 + 7 = 47$ Ausführungen des Schleifenkörpers gezählt, gegenüber 41 vor dem Loop Unrolling.

Mit zusätzlichen *flow*-Annotationen ist es möglich, dieses Problem zu verringern. Sei u der verwendete Unrolling-Faktor, n_{l1} die Ausführungsanzahl des Schleifenkörpers der abgerollten Schleife, n_{l2} die Ausführungsanzahl des Schleifenkörpers der zusätzlichen Schleife, n_u die Ausführungsanzahl der umgebenden Funktion oder Schleife und x die maximale Iterationsanzahl der ursprünglichen Schleife. Dann gilt die folgende lineare Ungleichung:

$$u \cdot n_{l1} + n_{l2} \leq x \cdot n_u$$

Die Werte n_{l1} und n_{l2} geben die Summen aller Ausführungen der jeweiligen Schleifenkörper über alle Aufrufe der umgebenden Funktion bzw. Iterationen der um-

gebenden Schleife an. Die Ausführungsanzahl des umgebenden Code n_u ist somit implizit enthalten.

Diese Ungleichung kann als *flow*-Annotation angegeben werden, um die Anzahl der gezählten Iterationen der zusätzlichen Schleife zu begrenzen. Für das oben angegebene Beispiel könnte die Annotation wie folgt aussehen:

```
flow sum 8 (0xa000026c) + (0xa00002a8) <= 41 (0xa0000244);
```

Die beiden ersten Adressen repräsentieren dabei die beiden Schleifen, die dritte Adresse eine Instruktion außerhalb der Schleifen. Allerdings wird die Ausführung der beiden Schleifen durch diese Annotation immer noch nicht exakt beschrieben, da die Ungleichung nur das Verhältnis der Summen der Ausführungen über alle Kontexte beschreibt. Die Pfad Analyse kann in vielen Kontexten weiterhin die maximale Iterationsanzahl für die zusätzliche Schleife zählen, indem weniger Iterationen der abgerollten Schleife gezählt werden, so dass in der Summe das Verhältnis erfüllt ist.

aiT unterstützt auch *flow*-Annotationen, die für jeden Kontext einzeln erfüllt werden müssen. Allerdings können diese hier nicht verwendet werden, da hier mehrere verschiedene Kontexte betroffen sind.

Ein Nachteil der zusätzlichen *flow*-Annotationen ist, dass dadurch die Analysezeit teilweise stark ansteigt. Bei einigen Benchmarks musste die Anzahl unterschiedener Kontexte reduziert werden, um eine Analyse in angemessener Zeit zu ermöglichen.

4.3.3 Durchgeführte Experimente

Das Loop Unrolling wurde für die fünf Benchmark-Programme FIR [Mär06], CRC [Mär06], G.721 (Encoder) [LPMS97], GSM [LPMS97] und SHA [GRE+01] untersucht. Wie bereits beim Procedure Cloning wurde die Optimierung manuell durchgeführt. Anhand des Quelltextes und der Ergebnisse einer WCET-Analyse der unoptimierten Version wurden die abzurollenden Schleifen ausgewählt. Für jeden Benchmark wurde eine Schleife, jeweils mit den Unrolling-Faktoren 2, 4 und 8, abgerollt.

Die betrachteten Benchmarks lassen sich in zwei Klassen unterteilen. Bei FIR, CRC und G.721 wurden Schleifen mit variabler Iterationsanzahl abgerollt. D.h. es wurde eine zusätzliche Schleife zur Behandlung der letzten Iterationen eingefügt und die oben beschriebenen *flow*-Annotationen generiert. Bei den Benchmarks GSM und SHA wurden dagegen Schleifen mit einer konstanten, und durch die verwendeten Unrolling-Faktoren teilbaren, Iterationsanzahl betrachtet.

Alle Benchmarks wurden wieder mit den höchsten Optimierungsstufen der verwendeten Compiler compiliert, und die WCET, ACET sowie die Codegröße vor und nach dem Loop Unrolling ermittelt. Für die TriCore Architektur wurden lediglich die Benchmarks FIR, G.721 und SHA betrachtet, da für die Benchmarks CRC und GSM die WCET-Analyse mit aiT für den TC1796 nicht erfolgreich durchgeführt werden konnte.

Die Länge der *call strings* bei der WCET-Analyse wurde bei allen Benchmarks auf 7 begrenzt und es wurden maximal 3 Kontexte pro Schleife unterschieden. Beim Benchmark FIR für den Thumb-Instruktionssatz des ARM7 musste bei den Unrolling-Faktoren 4 und 8 die Anzahl der Kontexte pro Schleife auf 1 herabgesetzt werden.

4.3.4 Ergebnisse

WCET und ACET

In den Abbildungen 4.15 bis 4.17 ist die relative WCET und ACET der Benchmarks nach dem Loop Unrolling dargestellt. Die WCETs bzw. ACETs der unoptimierten Versionen entsprechen 100%. Es wurden Reduzierungen der ACET bis zu $21,8\%$ (FIR/ARM/$u = 8$) erreicht. Für fast alle Benchmarks steigt die erreichte Verbesserung bei allen Architekturen mit dem verwendeten Unrolling-Faktor. Eine Ausnahme bildet der Benchmark G.721. Hier wird die ACET bei einem Unrolling-Faktor von 4 mit $7,8\%$ (ARM) zwar stärker reduziert als beim Unrolling-Faktor 2 ($3,2\%$ ARM), aber beim Unrolling-Faktor 8 reduziert sich die Verbesserung wieder etwas auf $5,7\%$ (ARM). Dies liegt an der geringen Iterationsanzahl der Schleife, die stets 7 oder 15 beträgt. Nach dem Abrollen um den Faktor 8 hat die Schleife nur noch höchstens eine Iteration, die Schleife zur Behandlung der letzten Iteration hat dagegen immer 7 Iterationen.

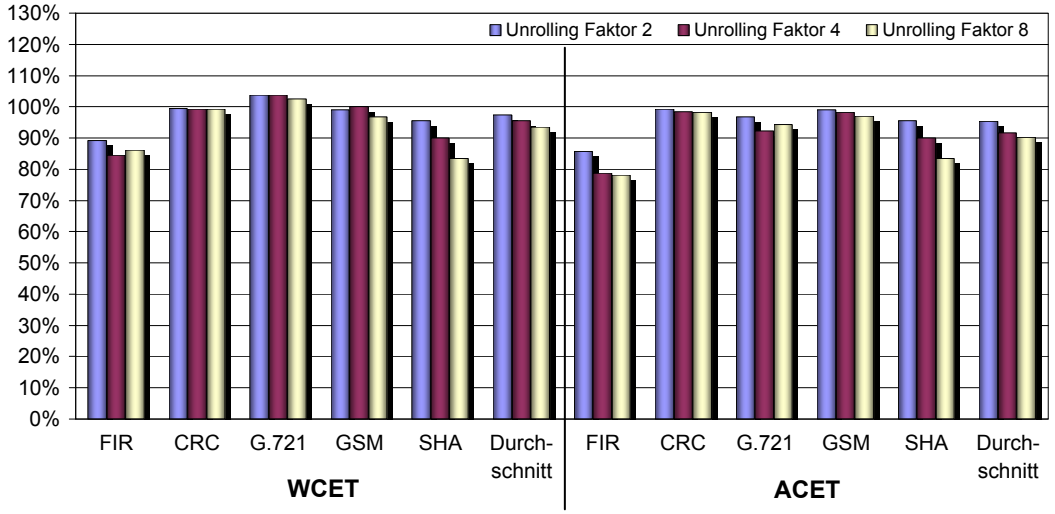

Abbildung 4.15: Relative WCET und ACET nach Loop Unrolling (ARM)

Auch beim Benchmark GSM ergibt sich eine leichte Verschlechterung der ACET um $1,7\%$ ($u = 2$) bzw. 2% ($u = 4$). Diese tritt allerdings nur beim ARM7 mit dem Thumb-Instruktionssatz auf. Diese Verschlechterung wird verursacht durch zusätzli-

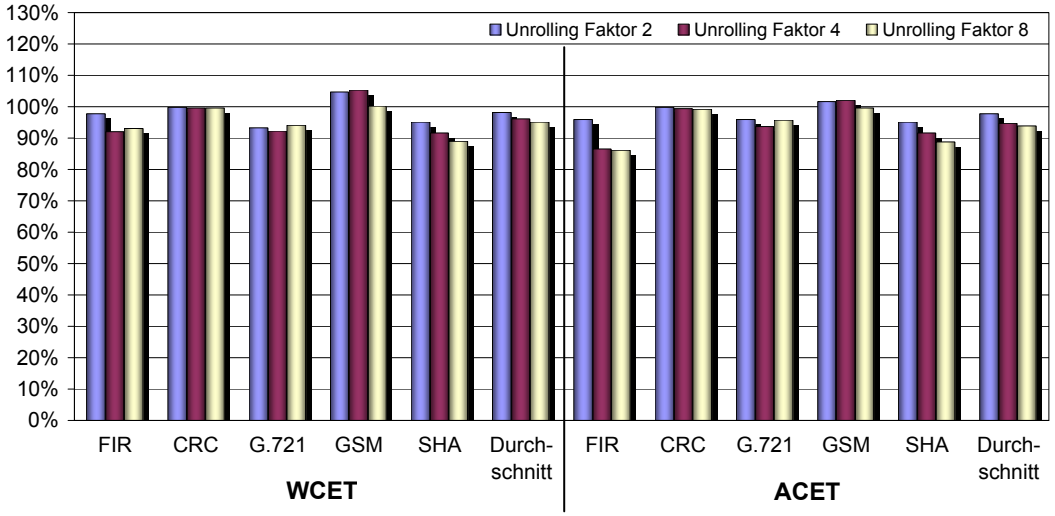

Abbildung 4.16: Relative WCET und ACET nach Loop Unrolling (Thumb)

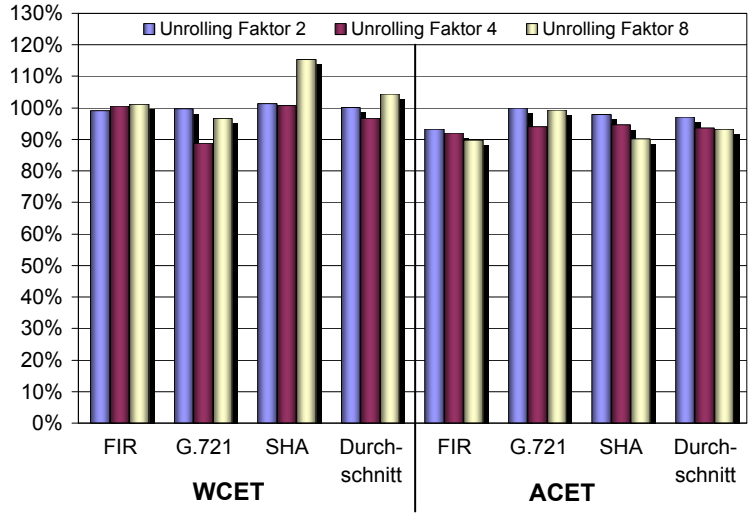

Abbildung 4.17: Relative WCET und ACET nach Loop Unrolling (TriCore)

che Speicherzugriffe, die aus einer ungünstigeren Registerbelegung resultieren, sowie durch zusätzliche Sprunginstruktionen, die notwendig werden, da die Reichweite von Sprüngen im Thumb-Modus sehr begrenzt ist.

Für die Benchmarks GSM und SHA, bei denen Schleifen mit konstanter Iterationsanzahl abgerollt wurden, verhalten sich die Änderungen der WCET ähnlich den Verbesserungen der ACET. Bei GSM betragen die Verbesserunegn der ACET für den ARM-Instruktionssatz des ARM7 $0,9\%$ ($u = 2$), $1,8\%$ ($u = 4$) und $3,1\%$ ($u = 8$). Die entsprechenden Verbesserungen der WCET liegen bei $0,9\%$ ($u = 2$), 0% ($u = 4$) und $3,2\%$ ($u = 8$). Beim Benchmark SHA sind die Werte sogar nahezu identisch. Der leicht schlechtere WCET-Wert bei GSM für den ARM und mit Unrolling-Faktor 4, ergibt sich durch ein, nur beim Unrolling-Faktor 4, vom Compiler nicht durchgeführtes Loop Invariant Code Motion. Auf die ACET hat dies keine Auswirkungen, da der Pfad in der Schleife, der die betroffenen Instruktionen enthält, für die verwendeten Eingabedaten nie ausgeführt wird.

Bei den Benchmarks CRC, FIR und G.721 sind die Verbesserungen der WCET einiges geringer als die der ACET. Während die Reduzierung der WCET für die Unrolling-Faktoren 2 und 4 meist noch zunimmt, sinkt sie beim Unrolling-Faktor 8 in fast allen Fällen wieder. Bei FIR für ARM-Instruktionssatz des ARM7 betragen die ACET-Verbesserungen beispielsweise $14,3\%$ ($u = 2$), $21,2\%$ ($u = 4$) und $21,8\%$ ($u = 8$), während die WCET-Verbesserungen bei nur $10,8\%$ ($u = 2$), $15,6\%$ ($u = 4$) und $13,9\%$ ($u = 8$) liegen. Lediglich bei CRC konnte durch zusätzliche *flow*-Annotationen auch für den Unrolling-Faktor 8 eine weitere (sehr geringe) Verbesserung der WCET-Reduzierung auf $0,84\%$ (ARM) erreicht werden, gegenüber $0,8$ (ARM) beim Unrolling-Faktor 4.

Die Verschlechterungen der WCET um $3,8\%$ ($u = 2$, $u = 4$) bzw. $2,5\%$ ($u = 8$) bei G.721 für den ARM resultieren aus der Struktur der Schleife, die über eine *return*-Anweisung verlassen werden kann. Der Compiler `armcc` erzeugt dadurch in den abgerollten Versionen der Schleife mehrere bedingte Rücksprungbefehle. aiT zählt allerdings für diese Instruktionen immer 5 Taktzyklen, auch wenn sie nicht ausgeführt werden.

Einfluss der Annotationen auf die WCET

Durch die angegebenen *flow*-Annotationen konnte die WCET bei FIR und CRC etwas reduziert werden. Abbildung 4.18 zeigt die relative WCET nach Loop Unrolling ohne Angabe der *flow*-Annotationen. Es ist jeweils die WCET nach der Optimierung relativ zur WCET vor der Optimierung dargestellt, welche 100% entspricht. Ohne die Annotationen liegt auch bei CRC die WCET beim Unrolling-Faktor 8 etwas höher als beim Faktor 4.

Beim Benchmark G.721 haben die Annotationen keine Auswirkung. Dies liegt daran, dass die maximale Iterationsanzahl der abgerollten Schleife 15 beträgt. Nach dem

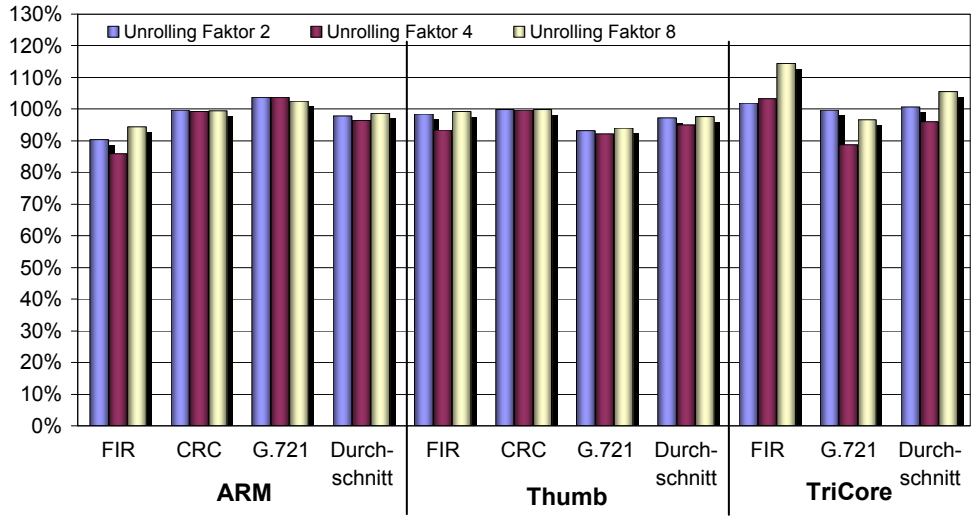

Abbildung 4.18: Relative WCET nach Loop Unrolling ohne flow Annotationen

Abrollen der Schleife um den Faktor 2, 4 oder 8 ergeben sich bei maximaler Iterationsanzahl der beiden erzeugten Schleifen immer 15 Ausführungen des ursprünglichen Schleifenkörpers. Es werden also nicht mehr Iterationen gezählt als vorher.

Codegröße

Abbildung 4.19 zeigt die relative Codegröße nach dem Loop Unrolling. Für die meisten Benchmarks sind die Zuwächse sehr gering. Lediglich bei CRC sind die Werte etwas höher, mit Vergrößerungen von $3,4\%$ (ARM/$u = 2$) bis $12,6\%$ (ARM/$u = 8$). Bei G.721 für den TriCore ist die Codegröße nach dem Loop Unrolling sogar geringer als vorher, da der verwendete Compiler `tricore_gcc` bei der unoptimierten Version Inlining für eine Funktion, die die abgerollte Schleife enthält, durchführt, nach dem Loop Unrolling aber nicht mehr.

4.4 Fazit

Mit den in diesem Kapitel vorgestellten Kontrollfluss-Optimierungen konnten teilweise erhebliche Reduzierungen der WCET erreicht werden, die teilweise sogar deutlich über den Verbesserungen der ACET lagen. Diese starken Verbesserungen der WCET wurden im Wesentlichen durch eine Reduzierung der WCET-Überschätzung erreicht.

Beim Loop Nest Splitting konnten mit Hilfe der während der Optimierung gewonnen Informationen über die Ausführungsanzahl der verschiedenen Pfade *flow*-Annotationen angegeben werden, die zu einer besseren Berücksichtigung des mögli-

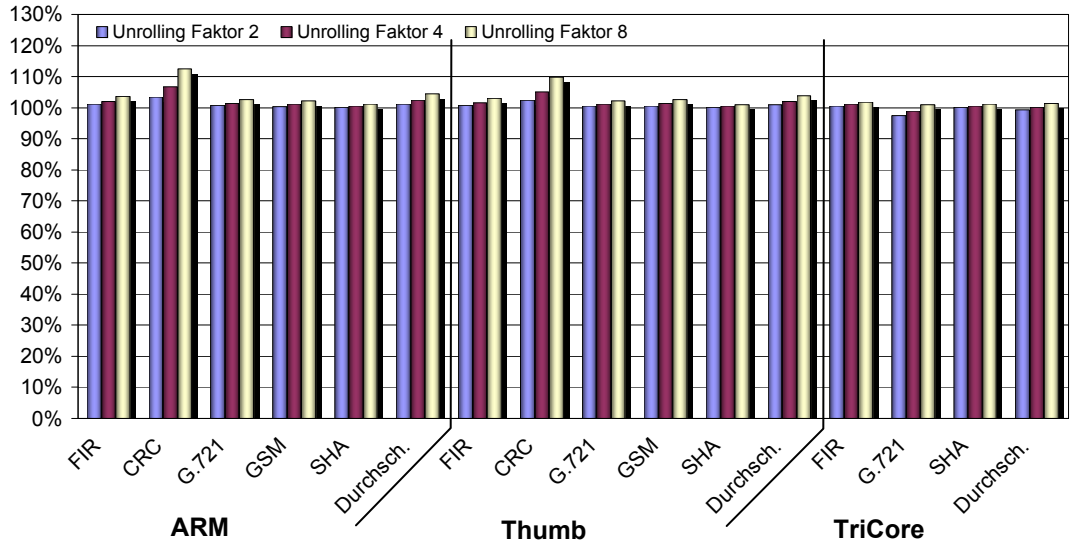

Abbildung 4.19: Relative Codegröße nach Loop Unrolling

chen Kontrollflusses bei der WCET-Analyse führten. Für einen Benchmark konnte die WCET um bis zu 89% reduziert werden, während die entsprechende ACET-Verbesserung nur 59, 4% betrug. Allerdings hängt der Erfolg der Optimierung auch stark von der Struktur eines Programms ab. Für die anderen beiden untersuchten Benchmarks waren die WCET-Verbesserungen bei allen Architekturen etwas niedriger als die entsprechenden ACET-Verbesserungen. Allerdings wurden sowohl bei der ACET als auch bei der WCET in allen Fällen Verbesserungen erreicht.

Die höchsten WCET-Verbesserungen von bis zu 95, 7% wurden durch das Procedure Cloning erreicht. Die Auswirkungen auf die ACET waren mit Verbesserungen von maximal 3% dagegen sehr gering. Die hohen WCET-Verbesserungen basieren im Wesentlichen auf der Reduzierung der WCET-Überschätzung durch eine genauere Angabe der maximalen Iterationsanzahl der Schleifen nach dem Procedure Cloning. Procedure Cloning ermöglicht es Kontrollfluss-Annotationen für die WCET-Analyse für verschiedene Aufrufe einer Funktion separat anzugeben. Dazu ist allerdings das Duplizieren von Code erforderlich, was zu einer Erhöhung der Codegröße führt. Da aiT durch das Unterscheiden von Kontexten verschiedene Aufrufe einer Funktion in den meisten Fällen bereits separat betrachtet, wäre zu prüfen ob unterschiedliche Annotationen für verschiedene Aufrufe einer Funktion nicht auf anderem Wege an aiT übergeben werden können, so dass das Klonen von Funktionen hierfür unnötig ist.

Bei der Betrachtung der Auswirkungen des Loop Unrolling auf die WCET müssen die beiden Fälle einer variablen Iterationsanzahl und einer konstanten Iterationsanzahl unterschieden werden. Bei einer variablen Iterationsanzahl muss eine zusätzliche Schleife zur Behandlung der letzten Iterationen eingefügt, die negative Auswirkun-

gen auf die WCET hat. Selbst nach der Angabe zusätzlicher *flow*-Annotationen waren die erreichten WCET-Verbesserungen geringer als die ACET-Verbesserungen. Da außerdem der Analyse-Aufwand durch die zusätzlichen Annotationen steigt, ist Loop Unrolling von Schleifen mit variabler Iterationsanzahl für einen WCET-optimierenden Compiler weniger geeignet. Bei den Schleifen mit konstanter Iterationsanzahl, bei denen keine zusätzliche Schleife nach dem Abrollen erforderlich war, lagen die WCET- und ACET-Verbesserungen etwa in derselben Größenordnung. Dies zeigt, dass die bei den Experimenten abgerollten Schleifen sowohl auf dem WC Pfad als auch auf dem betrachteten realen Ausführungspfad liegen. Könnte in einem Programm eine Schleife abgerollt werden, die auf dem WC Pfad liegt, bei realen Ausführungen aber nur selten ausgeführt wird, so wäre vermutlich auch eine WCET-Verbesserung möglich, die höher ist als die entsprechende ACET-Verbesserung.

5 Worst-Case Pfad Optimierung

Eine Möglichkeit die Ausführungszeit von Programmen zu optimieren besteht darin, speziell einen besonders häufig ausgeführten Pfad zu betrachten, und durch Codetransformationen die Ausführungszeit dieses Pfades zu reduzieren. Dabei wird oft in Kauf genommen, dass sich die Ausführungszeit anderer, weniger häufig ausgeführter Pfade erhöht. Traditionell wird solch ein Pfad anhand von Profiling-Daten ermittelt. D.h. es wird bei einer Ausführung des Programms mit üblichen Eingabedaten die Ausführungshäufigkeit der verschiedenen Pfade gemessen.

Der für die WCET-Analyse relevante Worst-Case Pfad muss jedoch nicht mit dem am häufigsten ausgeführten Pfad übereinstimmen. Tatsächlich gibt es oft sogar große Unterschiede. Ein Beispiel hierfür ist ein *if*-Block zur Behandlung eines Sonderfalles, der nur selten ausgeführt wird, und somit keinen großen Einfluss auf die Laufzeit des Programms hat. Dieser Sonderfall stellt aber den Worst-Case dar, d.h. der *if*-Block liegt auf dem WC Pfad und kann durchaus eine großen Einfluss auf die Gesamt-WCET haben.

Daraus folgt, dass die Optimierung eines häufig ausgeführten Pfades nicht die gleichen Auswirkungen auf die WCET haben muss, wie auf die durchschnittliche Laufzeit. Deshalb muss zur Reduzierung der WCET statt häufig ausgeführter Pfade der Worst-Case Pfad betrachtet werden.

Die Optimierung des WC Pfades ist allerdings komplizierter, denn eine Verbesserung des WC Pfades kann leicht dazu führen, dass ein anderer Pfad, der möglicherweise nur geringfügig kürzer ist, der neue WC Pfad wird. Weitere Optimierungen des ursprünglichen Pfades hätten somit keine Auswirkungen auf die WCET des Programms. Auch ist es möglich, dass nach einer Optimierung des WC Pfades, auf Kosten anderer Pfade, ein anderer Pfad länger ist als der WC Pfad vor der Optimierung, womit die Gesamt-WCET erhöht wurde.

Im folgenden Abschnitt wird die Optimierung des WC Pfades durch Code Positioning untersucht.

5.1 Code Positioning

Jedes Programm lässt sich in Basisblöcke unterteilen. Ein Basisblock ist eine Sequenz aufeinanderfolgender Instruktionen, die höchstens am Ende eine Sprunginstruktion enthält, und bei der lediglich die erste Instruktion ein Sprungziel seien

kann. Die Basisblöcke eines Programms lassen sich auf verschiedene Weisen im Speicher anordnen, so dass auf bestimmten Pfaden Sprünge notwendig sind, während auf anderen Pfaden die Basisblöcke direkt aufeinander folgen. Abbildung 5.1 zeigt ein Beispielprogramm für den ARM7, welches aus vier Basisblöcken besteht. Es gibt zwei mögliche Pfade von Basisblock 1 nach Basisblock 4: entweder über Basisblock 2 oder über Basisblock 3. Auf beiden Pfaden liegt jeweils ein Sprung (dargestellt durch eine gestrichelte Linie) und eine direkter Übergang zwischen zwei Basisblöcken. Sprünge während der Programmausführung führen zu Kontrollhazards und somit zu *pipeline stalls*. Dieser hat beim ARM7 immer eine Länge von zwei Taktzyklen. Ziel des Code Positioning ist es, die Basisblöcke eines Programms so anzuordnen, dass die Anzahl der durchgeführten Sprünge minimiert wird. Da hier die WCET reduziert werden soll, wird eine Anordnung gesucht, die die Anzahl der Sprünge auf dem Worst-Case Pfad minimiert.

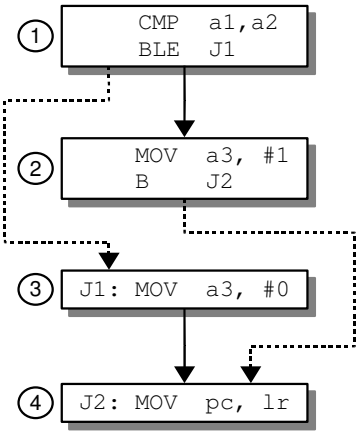

Abbildung 5.1: Beispiel Basisblöcke

Der in [ZWHM04] vorgestellte Code Positioning Algorithmus zur WCET-Optimierung verwendet ein Pfad-basiertes WCET-Analyse Werkzeug, welches in einen Compiler integriert ist. Hier werden zu Beginn alle Basisblöcke als unpositioniert betrachtet. Dazu werden am Ende aller Basisblöcke unbedingte Sprünge angehängt, falls diese nicht bereits mit einem unbedingten Sprung oder einer *return*-Instruktion enden. Auf diese Weise soll vermieden werden, dass durch das aufeinanderfolge Platzieren zweier Basisblöcke auf dem WC Pfad ein Sprung auf einem anderen Pfad nötig wird, wodurch sich der WC Pfad ändern könnte.

Nach dem Einfügen der unbedingten Sprünge wird in einem Greedy-Verfahren anhand der WCETs der Pfade jeweils ein Block ausgewählt und so platziert, dass auf dem WC Pfad kein Sprung durchgeführt wird. Nach jedem platzierten Block wird erneut die WCET aller Pfade berechnet.

In dieser Arbeit wurde hingegen das WCET-Analyse Werzeug aiT verwendet, das einen Worst-Case Pfad durch ein IPET-basiertes Verfahren berechnet. Das in

[ZWHM04] vorgestellte Verfahren kann hier nicht verwendet werden, da nicht die WCET aller Pfade explizit berechnet wird. Stattdessen wird durch das Lösen eines ganzzahligen Optimierungsproblems, unter Berücksichtigung der Kontrollstruktur sowie in vorherigen Analysephasen ermittelten Einschränkungen des möglichen Kontrollflusses, ein mögliches Ausführungsszenario ermittelt, welches zur maximalen Laufzeit führt. Als Ergebnis der WCET-Analyse steht neben der WCET aller Basisblöcke die Ausführungsanzahl aller Pfade in dem ermittelten Worst-Case Szenario zur Verfügung. Pfade, deren Ausführungsanzahl Null ist, liegen nicht auf dem WC Pfad. Somit ist durch diese Informationen implizit auch der WC Pfad gegeben. Auch die Ergebnisse eines IPET-basierten Verfahrens können für das Code Positioning verwendet werden.

5.1.1 Ablauf der Optimierung

Als erster Schritt wird eine WCET-Analyse des betrachteten Programms durchgeführt, um die WCET der Basisblöcke sowie die Ausführungsanzahl aller Pfade zur Durchführung der Optimierung zu erhalten. Anschließend wird jeweils ein kleiner Ausschnitt des Programms, bestehend aus einigen Basisblöcken, betrachtet, um eine möglichst günstige Anordnung dieser Blöcke zu bestimmen. Ein betrachteter Ausschnitt beginnt mit einem Basisblock, der auf dem WC Pfad liegt (d.h. eine Ausführungsanzahl größer als Null besitzt), und an dessen Ende sich der Kontrollfluss aufteilt. Der Ausschnitt endet mit dem Basisblock, an dem der Kontrollfluss wieder zusammenführt. Ein einfaches Beispiel für einen solchen Ausschnitt ist die in Abbildung 5.2 (a) dargestellte *if-then-else*-Struktur. Das Beispiel bezieht sich auf die ARM-Architektur. Die Werte an den Kanten geben die jeweilige Ausführungsanzahl an, und die Werte in den Blöcken die jeweilige WCET.

Es gibt nun vier Möglichkeiten, die Blöcke 2 und 3 anzuordnen. Bei der in Abbildung 5.2 (a) dargestellten Anordnung wird auf beiden möglichen Pfaden ein Sprung durchgeführt. Auf dem Pfad durch Block 3 werden, zusätzlich zur eigentlichen Ausführungszeit des Basisblockes 3, 2 Taktzyklen gezählt, um den durch den Sprung von Block 1 nach Block 3 verursachten *pipeline stall* zu berücksichtigen. Auf dem Pfad durch Block 2 sind es, durch die unbedingte Sprunginstruktion am Ende von Block 2, sogar 3 Taktzyklen. Alternativ könnte die Position der Blöcke vertauscht werden, wodurch sich die Ausführungszeit des Pfades durch Block 3 um 1 erhöht und die des Pfades durch Block 2 um 1 verringert, oder es könnte einer der Blöcke an das Ende der Funktion verschoben werden, wodurch auf einem der Pfade gar keine Sprünge mehr vorkommen.

Für das vorgestellte Beispiel ist die günstigste Möglichkeit das Verschieben des Blockes 3, der nur eine geringe Ausführungsanzahl hat, an das Ende der Funktion, wie in Abbildung 5.2 (b) dargestellt. Dadurch erhöht sich die Ausführungszeit des Pfades durch Block 3 um 3 Taktzyklen, die des anderen Pfades reduziert sich um 3 Taktzyklen. Damit ist der Pfad durch Block 2 aber weiterhin der Worst-Case Pfad,

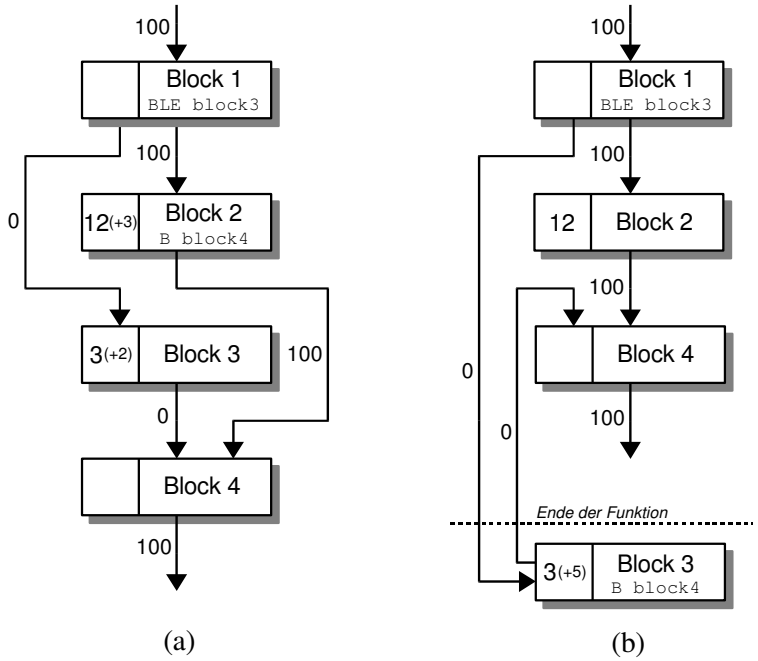

Abbildung 5.2: Beispiel Code Positioning

und die WCET des betrachteten Ausschnitts wurde um 3 Taktzyklen reduziert.

Insbesondere wenn sich die Ausführungszeiten der Blöcke nur gering unterscheiden, kann das Verschieben eines Blockes an das Ende der Funktion dazu führen, dass dieser Pfad anschließend länger ist als der ursprünglich längere Pfad. In diesem Fall muss eine der Möglichkeiten mit einem Sprung auf jedem Pfad gewählt werden.

Beim Thumb-Modus des ARM7TDMI muss bei der Positionierung zusätzlich beachtet werden, dass die Reichweite von Sprüngen sehr begrenzt ist. Das Verschieben eines Blockes an das Ende der Funktion ist mit einer einzelnen Sprunginstruktionen nicht immer möglich.

5.1.2 Durchgeführte Experimente

Das Code Positioning wurde für die Benchmarks ADPCM (Coder und Decoder), Motion Estimation, QSDPCM und Cavity durchgeführt. Dazu wurde zunächst der C-Quelltext der Programme compiliert und Assemblercode erzeugt. In dem erzeugten Assemblercode wurden anschließend einzelne Basisblöcke, wie oben beschrieben, verschoben. Dabei wurden ggf. Sprunginstruktionen eingefügt, entfernt, oder die Bedingung invertiert.

Der Assemblercode vor und nach der Optimierung wurde anschließend assembliert und als ausführbare ELF-Datei gelinkt. Für diese wurde mit aiT eine WCET-

Abschätzung berechnet und durch Simulation eine ACET bestimmt.

Für den Benchmarks ADPCM wurde die Länge der *call strings* bei der WCET-Analyse auf 7 begrenzt und es wurden 3 Kontexte pro Schleife unterschieden. Für die Benchmarks ME, QSDPCM und Cavity wurde die Länge der *call strings* auf 5 begrenzt und es wurden 2 Kontexte pro Schleife unterschieden.

Das Code Positioning wurde nur für den ARM7TDMI im ARM- bzw. Thumb-Modus durchgeführt. Beim TC1796 sind die Auswirkungen des Verschiebens eines Basisblockes durch die Komplexität des Prozessors sehr viel komplizierter und können kaum lokal betrachtet werden. Der TC1796 besitzt beispielsweise einen 8 Byte Instruktionspuffer. Es werden immer 8 Bytes (also zwei bis vier Instruktionen) gleichzeitig in den Puffer geladen und anschließend an die verschiedenen Pipelines weitergeleitet. Wird das Programm aus dem Flash-Speicher des TC1796 ausgeführt, benötigt das Laden neuer Instruktionen einige Taktzyklen. Bei einem Sprung muss der Puffer neu gefüllt werden. Die hierfür nötigen Taktzyklen müssten beim Verschieben eines Basisblockes berücksichtigt werden. Auch die Ausrichtung der Instruktionen und die Größe eines Basisblocks haben somit einen Einfluss auf die Änderungen der WCET beim Verschieben eines Blockes.

Weitere Faktoren, die bei der Platzierung von Basisblöcken berücksichtigt werden müssen, sind die statische Sprungvorhersage des TC1796 sowie der Instruktionscache, falls dieser verwendet wird.

Die Diagramme, in denen aiT den genauen Prozessorzustand zu jedem Taktzyklus darstellt, sind für den TriCore sehr komplex. Die Anzahl der möglichen Zustände ist selbst für kleinere Programme oft enorm hoch und die Beschreibung des Prozessorstatus, ohne genaue Kenntnisse über die interne Funktionsweise des Prozessors, schwer zu verstehen. Dadurch ist es sehr schwierig die verschiedenen Auswirkungen des Verschiebens eines Basisblockes anhand dieser Diagramme nachzuvollziehen.

5.1.3 Ergebnisse

In Abbildung 5.3 ist die relative WCET und ACET nach der Durchführung des Code Positioning dargestellt. Die Reduzierung der ACET ist für alle durchgeführten Tests niedriger als die der WCET und liegt zwischen 0% (Cavity, ADPCM Dec./ARM) und $5,6\%$ (QSDPCM/ARM). Bei Motion Estimation ist die ACET um $0,88\%$ (ARM) bzw. $0,52\%$ (Thumb) gestiegen. Es hat sich also bestätigt, dass eine Optimierung des Worst-Case Pfades nicht unbedingt positive Auswirkungen auf die ACET haben muss.

Die Verbesserungen der WCET liegen für den ARM-Instruktionssatz zwischen $8,4\%$ (ADPCM Coder) und $1,1\%$ (ME). Bei Cavity konnte keine Verbesserung erreicht werden. Für den Thumb-Instruktionssatz sind die WCET-Verbesserungen etwas geringer. Sie liegen zwischen $3,8\%$ (ADPCM Coder) und $0,2\%$ (Cavity). Die durchschnittliche Verbesserung der WCET über alle Benchmarks beträgt für den

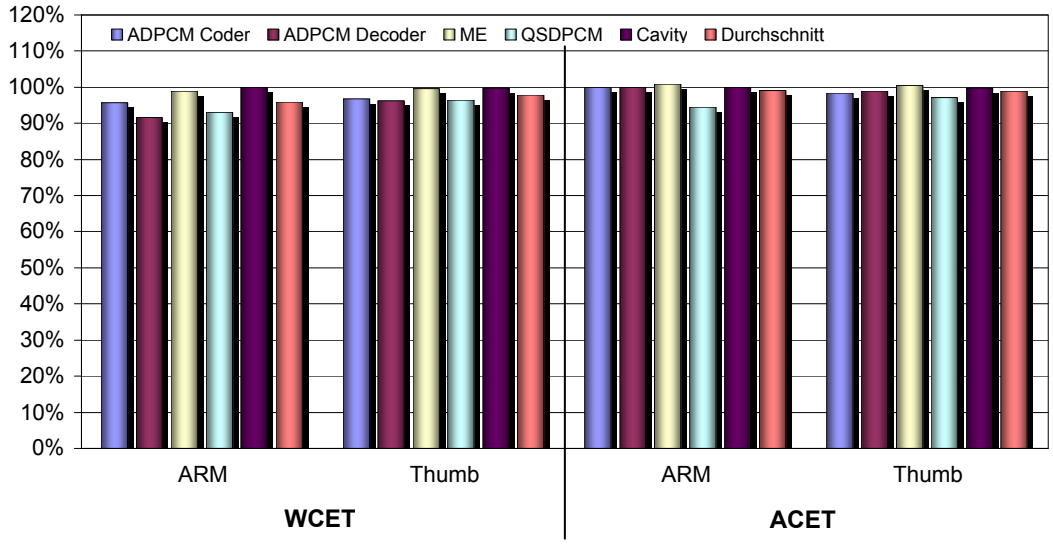

Abbildung 5.3: Relative WCET und ACET nach Code Positioning

ARM-Instruktionssatz $4,1\%$ und für den Thumb-Instruktionssatz $2,2\%$.

Es konnten bei den betrachteten Benchmarks jeweils nur sehr wenige Basisblöcke verschoben werden. Bei Cavity konnte für den ARM-Instruktionssatz kein einziger Block verschoben werden, obwohl dieser Benchmark viele *if*-Anweisungen enthält und somit auch viele Basisblöcke. Bei ADPCM Coder/Decoder für den ARM-Instruktionssatz wurde nur jeweils ein Basisblock verschoben. Da diese Blöcke in einer Schleife mit vielen Iterationen liegen, sind die Auswirkungen auf die WCET trotzdem recht hoch.

Ein Grund für die wenigen Optimierungsmöglichkeiten liegt darin, dass bei vielen Verzweigungen lediglich ein oder mehrere Basisblöcke übersprungen werden, wie bei einer *if-then*-Struktur ohne *else*-Teil. Diese *then*-Blöcke liegen in den meisten Fällen auf dem WC Pfad, und sie sind üblicherweise so angeordnet, dass bei der Ausführung keine Sprünge erforderlich sind. Im ARM-Modus können solche Strukturen auch dadurch entstehen, dass kurze *then*- oder *else*-Pfade einer *if*-Anweisung durch die bedingte Ausführung der Instruktionen realisiert werden, und nicht als eigener Basisblock.

Desweiteren konnten insbesondere im Thumb-Modus viele der *if-then-else*-Strukturen, wie in Abbildung 5.2 dargestellt, nicht optimiert werden, da beide Blöcke die gleiche oder eine ähnliche Ausführungszeit haben.

Die in [ZWHM04] erreichten WCET-Reduzierungen liegen für neun untersuchte Benchmarks zwischen $28,2\%$ und $1,6\%$. Die durchschnittliche WCET-Verbesserung ist mit $10,5\%$ etwas höher als die hier erreichten Verbesserungen.

5.2 Fazit

Durch die Optimierung des Worst-Case Pfades mittels Code Positioning konnten Verbesserungen der WCET von bis zu $8,4\%$ erreicht werden. Damit wurden bereits deutliche Verbesserungen erreicht, obwohl bei den betrachteten Benchmarks jeweils nur sehr wenige Basisblöcke vorschoben wurden.

Gleichzeitig waren die Verbesserungen der ACET nach der Optimierung nur sehr gering, und in einem Fall hat sich die ACET sogar verschlechtert. Dies zeigt, dass sich der WC Pfad der untersuchten Benchmarks vom Ausführungspfad bei den durchgeführten Tests unterscheidet. Das Code Positioning auf Basis von WCET-Informationen scheint somit gut geeignet für die Integration in einen WCET-optimierenden Compiler.

6 Standard Optimierungen

In den vorangegangenen Kapiteln wurden vier Compiler-Optimierungen detailliert bezüglich ihres Einflusses auf die WCET untersucht. Diese Optimierungen wurden manuell auf dem C-Quelltext (Kapitel 4) bzw. Assemblercode (Kapitel 5) ausgewählter Benchmarks durchgeführt. Es wurden jeweils nur die Auswirkungen einer einzigen Optimierung betrachtet.

In diesem Kapitel werden hingegen die Auswirkungen der in der Optimierungsphase einer existierenden Compilerumgebung durchgeführten Standard Optimierungen auf die WCET untersucht. Neben den einzelnen Optimierungen werden auch die gesamten Optimierungssequenzen untersucht, die in der Compilerumgebung während der verschiedenen Optimierungsstufen durchgeführt werden.

6.1 ICD-C Compiler Framework

Das ICD-C Compiler Framework [ICD06] ist ein am Informatik Centrum Dortmund entwickeltes Compiler-Frontend für die Programmiersprache C. Es überführt C-Code in eine High-Level Zwischendarstellung, in der sämtliche C-Konstrukte erhalten bleiben. Mit Hilfe einer zur Verfügung gestellten C++-Klassenbibliothek kann die Zwischendarstellung analysiert und manipuliert werden. Die integrierten Kontrollfluss-, Datenfluss- und Funktionsaufrufanalysen ermöglichen die Implementierung eigener, plattform-unabhängiger Codeanalysen oder -optimierungen, basierend auf der Zwischendarstellung.

Im Framework enthalten ist auch eine Optimierungsklasse, die viele übliche Standard Compiler-Optimierungen enthält und eine Optimierung der Zwischendarstellung mit drei verschiedenen Optimierungstufen ermöglicht.

Die erzeugte Zwischendarstellung kann über eine spezielle Schnittstelle an ein Compiler-Backend übergeben werden, welches Assemblercode für eine bestimmte Zielarchitektur generiert. Auf diese Weise können ganze Compiler Toolchains basierend auf dem ICD-C Framework erstellt werden. Alternativ kann die Zwischendarstellung aber auch wieder als C-Code ausgegeben werden, der anschließend mit einem beliebigen C-Compiler in Assemblercode übersetzt werden kann. Diese Möglichkeit kann beispielsweise zur Implementierung plattform-unabhängiger Optimierungen auf Quelltext-Ebene (C2C) verwendet werden.

Auch bei den hier durchgeführten Experimenten wurde die Möglichkeit genutzt,

die Zwischendarstellung wieder als C-Code auszugeben. So konnten die von ICD-C unterstützten Optimierungen auf dem C-Quelltext verschiedener Benchmarks angewandt werden, und die resultierenden C-Dateien mit den bereits in den vorherigen Kapitel verwendeten Compilern für die verschiedenen betrachteten Architekturen compiliert werden.

6.1.1 Optimierungen

Die ICD-C Optimierungsklassen enthalten 21 verschiedene Optimierungen, die auf der Zwischendarstellung durchgeführt werden können:

- **Fold Constant Code**
  Berechnungen, die nur Konstanten enthalten, werden ausgewertet und durch das Ergebnis ersetzt. Der Wert einer als konstant deklarierten Variablen wird an die Stellen kopiert, an denen die Variable verwendet wird. Desweiteren werden *if*-Anweisungen, deren Bedingungen nur Konstanten enthalten, ausgewertet und durch den *then*- bzw. *else*-Pfad ersetzt, Schleifen, deren Bedingungen nur Konstanten enthalten und als *false* ausgewertet werden, entfernt, sowie *switch*-Anweisungen mit konstantem Auswahl-Ausdruck durch den entsprechenden Code ersetzt.

- **Dead Code Elimination**
  Es werden Anweisungen, die durch keinen Kontrollflusspfad erreicht werden, entfernt. Aufeinanderfolgende Basisblöcke werden zusammengefasst, falls der Vorgänger-Block nur eine ausgehende Kontrollflusskante besitzt und der Nachfolge-Block nur eine eingehende Kante. Anweisungen, deren Ausführung keine Auswirkungen hat, sowie Zuweisungen, deren Ergebnis nie verwendet wird, werden entfernt.

- **Remove Unused Symbols**
  Bezeichner, die nie verwendet werden, werden entfernt. Dabei werden auch Funktionen entfernt, die nie aufgerufen werden.

- **Remove Unused Function Arguments**
  Funktionsparameter, deren Werte nie verwendet werden, werden entfernt.

- **Merge String Constant Expressions**
  Identische, konstante Zeichenketten, die mehrfach vorkommen, werden durch eine einzige Zeichenkette ersetzt.

- **Value Propagation**
  Wird einer Variablen x der Wert einer anderen Variablen y oder ein konstanter Integer-Wert zugewiesen, so kann x in folgenden Berechnungen durch y bzw. den Integer-Wert ersetzt werden, falls y zwischen der Zuweisung und

der Verwendung nicht verändert wird, und die Zuweisung die einzige mögliche Definition von x bei der betrachteten Verwendung ist.

- **Expression Simplification**
  Ausdrücke werden vereinfacht, indem z.B. Pre-Inkrement/Dekrement-Operatoren anstelle von Post-Inkrement/Dekrement-Operatoren verwendet werden, oder redundante Casts entfernt werden.

- **Transform Head Controlled Loops**
  Schleifen, bei denen die Abbruchbedingung zu Beginn der Schleife überprüft wird, wie z.B. *for*- und *while*-Schleifen, werden in *do-while*-Schleifen transformiert, bei denen der Test der Abbruchbedingung am Ende erfolgt.

- **Loop De-Indexing**
  Arrayzugriffe innerhalb von Schleifen werden durch automatisch inkrementierende Load/Store-Anweisungen ersetzt.

- **Loop Collapsing**
  Verschachtelte Schleifen, die mehrdimensionale Arrays bearbeiten, werden ersetzt durch einfache Schleifen, in denen das Array als eindimensional betrachtet wird.

- **Loop Unswitching**
  Enthält eine Schleife ausschließlich eine *if*-Anweisung oder eine *switch*-Anweisung, deren Bedingung unabhängig von der Schleifenvariablen ist, so wird diese vor die Schleife verschoben, und es wird für jeden möglichen Pfad eine eigene Schleife erzeugt.

- **Loop Unrolling**
  Schleifen werden um einen automatisch ermittelten Unrolling-Faktor u abgerollt, indem der Schleifenkörper mehrfach kopiert wird und die Iterationsanzahl der Schleife durch das Inkrementieren der Schleifenvariablen um u bei jeder Iteration angepasst wird. (vgl. Kapitel 4.3)

- **Tail Recursion Elimination**
  Bei rekursiven Funktionen, die den rekursiven Aufruf als letzte Anweisung enthalten, kann die Rekursion entfernt werden, indem der Aufruf durch einen Sprung an den Beginn der Funktion ersetzt wird.

- **Create Multiple Exits**
  Bei Funktionen, die eine einzige *return*-Anweisung enthalten, die über verschiedene Kontrollflusspfade erreicht wird, werden zusätzliche *return*-Anweisungen in den verschiedenen Pfaden eingefügt, um Sprünge zu vermeiden.

- **Inline Expansion**
  Für Funktionen, die eine bestimmte Größe nicht überschreiten, werden sämtliche Aufrufe der Funktion durch eine Kopie der Funktion selbst ersetzt.

- **Function Specialization**
  Für kleine Funktionen, die mit konstanten Parametern aufgerufen werden, werden spezialisierte Versionen erzeugt, bei denen bestimmte Parameter entfallen und durch Konstanten ersetzt werden. (vgl. Kapitel 4.2)

- **Eliminate Return Value**
  Wird der von einer Funktion zurückgegebene Wert bei keinem Aufruf der Funktion verwendet, so wird die Rückgabe des Wertes aus der Funktion entfernt.

- **Struct Scalarization**
  Es werden *struct*-Objekte, die Bitfelder enthalten, in äquivalente Integer-Objekte umgewandelt, und skalare Komponenten eines *struct*-Objektes durch eigenständige Objekte ersetzt.

- **Separate Life Ranges**
  Lokale Variablen, die in verschiedenen Zeitbereichen für unterschiedliche Zwecke genutzt werden, werden durch mehrere einzelne Variablen ersetzt.

- **Common Subexpression Elimination**
  Wird ein bestimmter Teilausdruck wiederholt an verschiedenen Stellen berechnet, so können diese Berechnungen durch eine einzige ersetzt werden, deren Ergebnis zwischengespeichert und anstelle einer erneuten Berechnung verwendet wird.

- **Redundant Load Elimination**
  Mehrere redundante Lesezugriffe auf globale Variablen werden vermieden, indem eine lokale Variable eingeführt wird, die den Wert der globalen Variablen zwischenspeichert.

6.1.2 Ablauf der Optimierung

Die ICD-C Optimierungsklasse enthält eine Funktion zur Optimierung der Zwischendarstellung, die alle hier aufgelisteten Optimierungen in einer sinnvollen Reihenfolge ausführt. Dabei werden drei verschiedene Optimierungsstufen unterstützt. In der Optimierungsstufe 1 werden nur einige grundlegende Optimierungen, wie beispielsweise *Constant Propagation*, *Copy Propagation* und *Dead Code Elimination* durchgeführt. In der Optimierungsstufe 2 werden die meisten der vorgestellten Optimierungen durchgeführt. Lediglich die drei Optimierungen *Inline Expansion*, *Loop Unrolling* und *Function Specialization*, die durch das Duplizieren von Code die Codegröße erhöhen, werden erst in der Optimierungsstufe 3 durchgeführt. In Abbildung 6.1 ist der genaue Ablauf der Optimierung dargestellt. Die unterschiedlichen Schattierungen geben an, ab welcher Optimierungsstufe eine Optimierung durchgeführt wird.

Der Ablauf der Optimierung lässt sich in mehrere Phasen unterteilen. In der initialen Phase werden nacheinander fünf verschiedene Optimierungen durchgeführt.

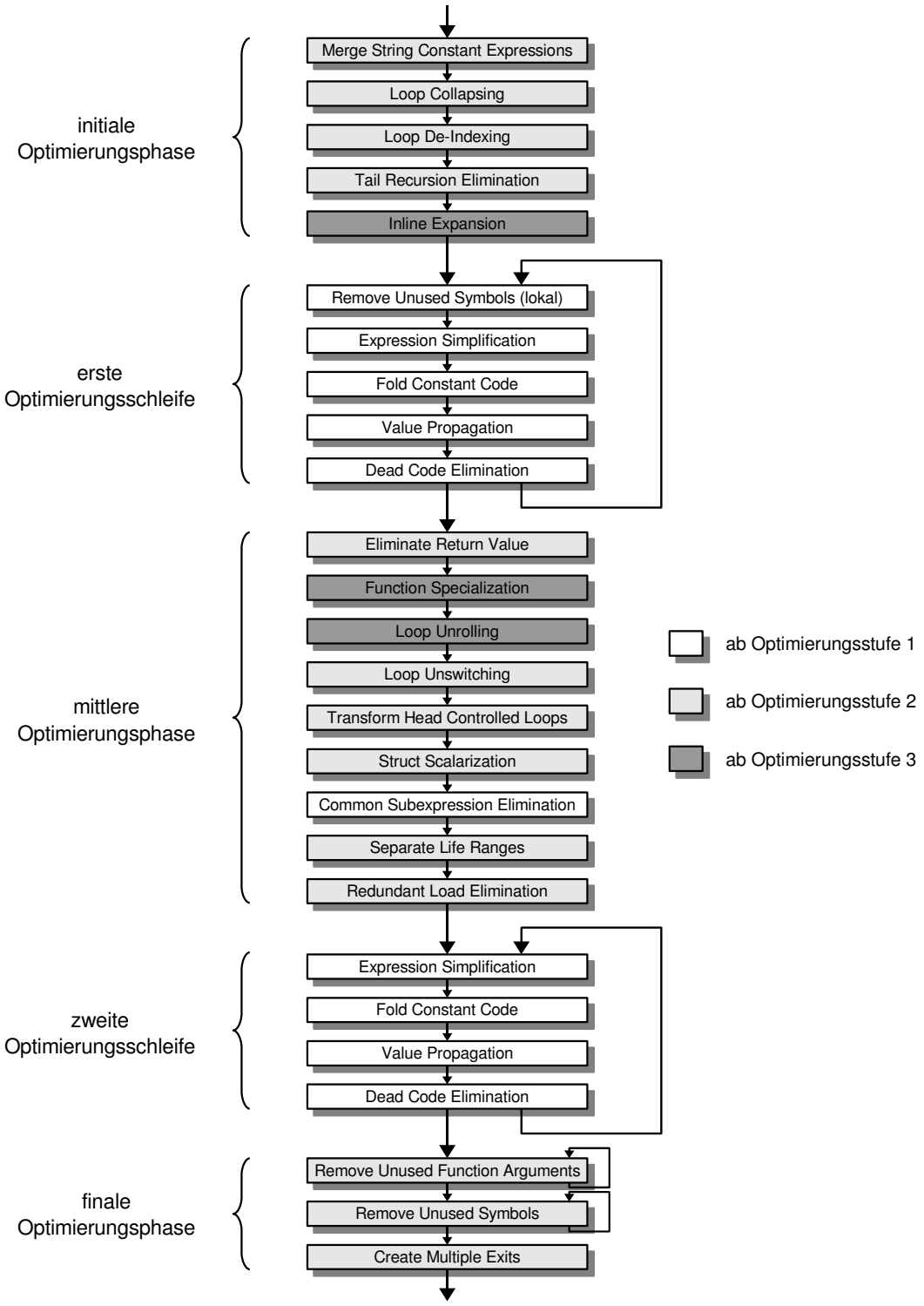

Abbildung 6.1: Ablauf der Optimierung in ICD-C

Danach folgt die erste Optimierungs-Schleife, bestehend aus den Optimierungen *Remove Unused Symbols* (lokal), *Expression Simplification*, *Fold Constant Code*, *Value Propagation* und *Dead Code Elimination*. Diese Schleife wird solange ausgeführt, bis keine Verbesserungen mehr möglich sind. Es folgt die mittlere Optimierungsphase, in der neun weitere Optimierungen nacheinander durchgeführt werden. Anschließend folgt eine zweite Optimierungsschleife, mit den vier Optimierungen *Expression Simplification*, *Fold Constant Code*, *Value Propagation* und *Dead Code Elimination*, die bereits in der ersten Schleife durchgeführt wurden. In einer abschließenden Phase werden die drei letzten Optimierungen Remove *Unused Function Arguments*, *Remove Unused Symbols* und *Create Multiple Exits* durchgeführt. *Remove Unused Function Arguments* und *Remove Unused Symbols* werden jeweils solange wiederholt, bis keine Verbesserungen mehr möglich sind.

Der beschriebene Ablauf bezieht sich auf die Optimierungsstufe 3, bei den niedrigeren Optimierungsstufen werden einige der Optimierungen übersprungen.

6.2 Durchgeführte Experimente

Die Auswirkungen der vorgestellten Optimierungen auf die WCET wurde an den acht Benchmarks ADPCM (Coder und Decoder), Bubblesort, CRC, FIR, G.721 (Coder und Decoder), GSM, Motion Estimation und SHA untersucht. Da die Optimierungen automatisch durchgeführt werden, konnte eine relativ große Anzahl an Optimierungen betrachtet werden. Da hier viele verschiedene Optimierungen gleichzeitig untersucht werden, mussten die verwendeten Benchmarks keinen besonderen Kriterien wie bei den Optimierungen in den Kapiteln 4 und 5 genügen, sondern konnten frei gewählt werden.

Zur Durchführung der Optimierungen auf den verschiedenen Benchmarks wurde das zum ICD-C Compiler Framework gehörende Programm `irinfo` verwendet. Mit Hilfe dieses Programms, welches auf den ICD-C Bibliotheken basiert, wurde der in Abschnitt 6.1.2 beschriebene Optimierungsablauf auf dem C-Quelltext aller untersuchten Benchmarks durchgeführt. Dieser Optimierungsschritt wurde für jeden Benchmark viermal durchgeführt, mit den Optimierungsstufen O0, O1, O2 und O3. Bei der Optimierungsstufe O0 werden keine Optimierungen durchgeführt. Die Ergebnisse dieser Optimierungsstufe wurden als Vergleichswerte für die anderen Optimierungsstufen verwendet.

Neben den drei Optimierungsabfolgen wurde auch jede Optimierung separat betrachtet. Dazu wurde für jede der Optimierungen eine spezielle Version des Programms `irinfo` erstellt, so dass ausschließlich diese eine Optimierung durchgeführt wird. Mit jeder dieser Versionen wurde der C-Quelltext aller Benchmarks transformiert.

Nach der Transformation der Benchmarks mit `irinfo` wurden die 26 unterschiedlichen Versionen jedes Benchmarks mit den Compilern `armcc/tcc` bzw. `tricore-gcc`

compiliert. Im Gegensatz zu den vorherigen Kapiteln wurden die Compiler hier mit dem Parameter `-O0` aufgerufen, d.h. es wurden keine weiteren Optimierungen durch die Compiler durchgeführt. Da die Optimierungen der Compiler sehr viel mächtiger sind als die mit ICD-C durchgeführten Optimierungen auf Quelltext-Ebene, hätten höhere Optimierungsstufen bei der Compilierung zur Folge, dass viele der ICD-C Optimierung nur sehr geringe oder gar keine Auswirkungen auf den generierten Code haben.

Für alle erzeugten ausführbaren Dateien wurde anschließend mit aiT eine WCET-Abschätzung berechnet und eine ACET im Prozessorsimulator ermittelt.

6.3 Anpassung der Annotationen

Bei einigen Optimierungen ist eine Anpassung der Annotationen für die WCET-Analyse der optimierten Version erforderlich. Dabei geht es hauptsächlich um die Anpassung der Loop Bound Annotationen. Beim *Loop Unrolling* wird beispielsweise die Anzahl der Iterationen verändert. Es hat sich gezeigt, dass die Loop Bound Analyse von aiT in keinem Fall die Loop Bounds einer abgerollten Schleife ermitteln konnte, auch wenn es für die ursprüngliche Schleife noch möglich war.

Bei *Inline Expansion* und *Function Specialization* können mehrere Kopien von Schleifen erzeugt werden, für die dann jeweils eigene Loop Bound Annotationen angegeben werden müssen.

Nach der Optimierung *Transform Head Controlled Loops* ist meist eine Anpassung vieler Loop Bound Annotationen erforderlich. In jeder Loop Bound Annotation muss durch eines der Schlüsselwörter `begin` oder `end` angegeben werden, ob sich der Test der Abbruchbedingung am Anfang oder am Ende der Schleife befindet, damit aiT die korrekte Iterationsanzahl berücksichtigt. Nach der Transformation befindet sich der Test der Abbruchbedingung immer am Ende der Schleife, und die Annotationen müssen ggf. angepasst werden. Es hat sich auch gezeigt, dass die Loop Bound Analyse von aiT für den ARM7 bei Schleifen, deren Abbruchbedingung am Ende des Schleifenkörpers geprüft wird, viel häufiger erfolgreich ist. Bei aiT für den TC1796 ist die Situation hingegen genau umgekehrt.

6.4 Ergebnisse

In den Abbildungen 6.2 bis 6.5 ist die relative WCET und ACET der Benchmarks, nach der Optimierung durch ICD-C mit den Optimierungsstufen O1, O2 und O3 dargestellt. Die Werte sind als Prozentzahl der unoptimierten Versionen (d.h. ICD-C Optimierungsstufe O0) angegeben, die 100% entsprechen.

Die Auswirkungen der Optimierungsstufe O1 sind bei den meisten Benchmarks sehr gering. Bei ADPCM, FIR, G.721, GSM und ME wurden Verbesserungen und

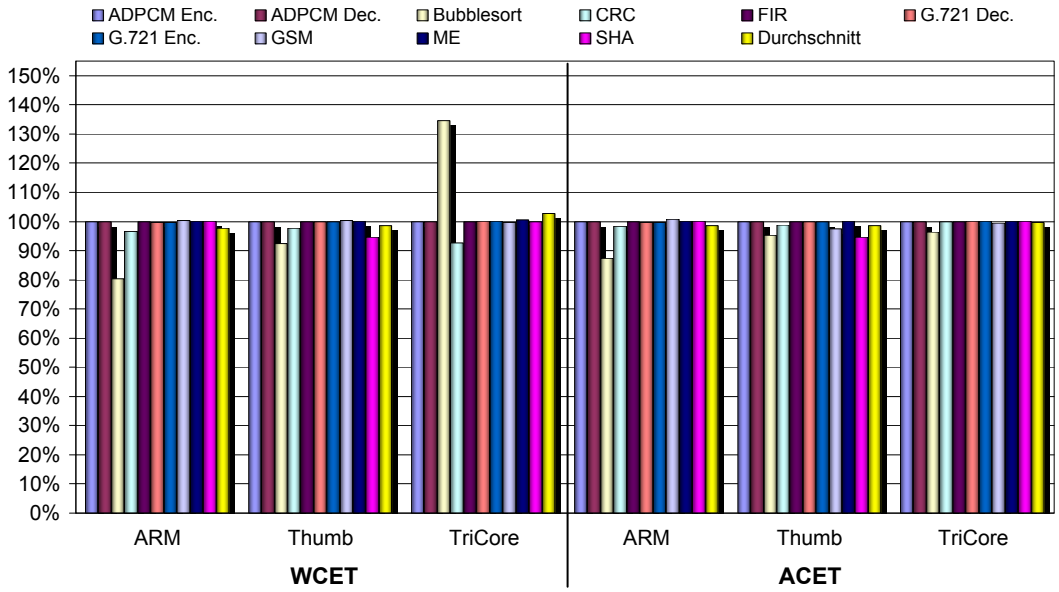

Abbildung 6.2: Relative WCET und ACET nach ICD-C Optimierung O1

teilweise auch Verschlechterungen sowohl der WCET als auch der ACET unter $0,5\%$ ermittelt. Bei SHA konnte für den ARM7 mit Thumb-Instruktionssatz eine Verbesserung der WCET und der ACET von $5,3\%$ erreicht werden. Die größten Reduzierungen wurden bei Bubblesort für den ARM7 mit ARM-Instruktionssatz erreicht, mit $19,5\%$ bei der WCET und $12,7\%$ bei der ACET.

In fast allen Fällen sind die Änderungen bei der WCET und der ACET sehr ähnlich. Eine Ausnahme bildet der Benchmark Bubblesort. Die Reduzierung der ACET liegt hier zwischen $12,7\%$ (ARM) und $3,5\%$ (TriCore). Beim ARM7 sind die WCET-Werte ähnlich, mit $19,5\%$ (ARM) und $7,7\%$ (Thumb). Die WCET-Reduzierung ist hier etwas höher als die ACET-Reduzierung, da die Optimierungen innerhalb eines *if*-Blockes in der innersten Schleife durchgeführt wurden, der für alle Iterationen auf dem WC Pfad liegt, aber bei einer Ausführung des Programms weniger häufig ausgeführt wird. Bei Bubblesort für den TriCore verschlechterte sich die WCET um $34,6\%$. Durch eine *Common Subexpression Elimination* werden nach der Optimierung mehr Speicherzugriffe generiert. Da die WCET für Speicherzugriffe in aiT für den TC1796 sehr hoch ist, kommt es zu der Verschlechterung der Gesamt-WCET. Da der verwendete Simulator für den TriCore für Speicherzugriffe meist nur einen Taktzyklus zählt, konnte die ACET trotzdem verbessert werden.

Die durchschnittliche Reduzierung der WCET über alle Benchmarks beträgt bei der Optimierungsstufe O1 $2,3\%$ (ARM) bzw. $1,5\%$ (Thumb). Beim TriCore ergibt sich im Durchschnitt eine Verschlechterung von $2,8\%$. Dieser Wert kommt durch die hohe Verschlechterung der WCET bei Bubblesort zu stande. Die durchschnittliche ACET-Verbesserung beträgt $1,4\%$ (ARM, Thumb) bzw. $0,4\%$ (TriCore).

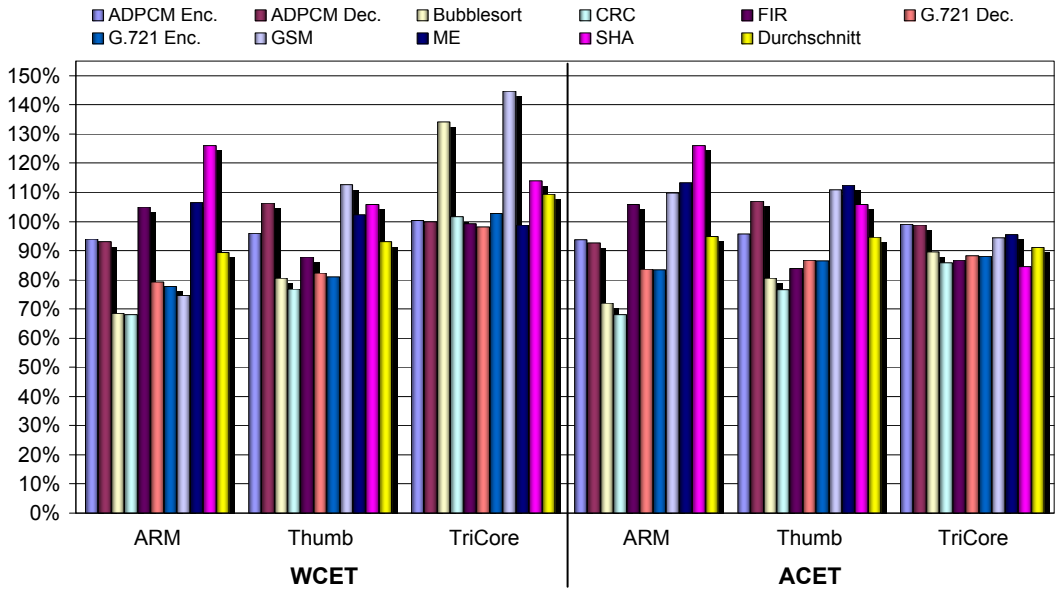

Abbildung 6.3: Relative WCET und ACET nach ICD-C Optimierung O2

Bei der Optimierungsstufe O2 wurden deutlichere Auswirkungen der Optimierung gemessen. Beim ARM7 mit ARM-Instruktionssatz wurden WCET-Reduzierungen bis zu $31,9\%$ (CRC) erreicht, beim Thumb-Instruktionssatz bis zu $23,1\%$ (CRC). Die maximale WCET-Reduzierung beim TriCore beträgt dagegen nur $1,9\%$ (G.721 Decode). Beim ARM7 wurden auch die höchsten ACET-Verbesserungen bei CRC erreicht, mit $31,9\%$ (ARM) und $23,4\%$ (Thumb). Die ACET konnte auch beim TriCore um bis zu $15,5\%$ (SHA) reduziert werden.

Für einige Benchmarks wurde aber durch die Optimierung sowohl die WCET als auch die ACET zum Teil deutlich verschlechtert. Bei GSM für den TriCore ist die WCET nach der Optimierung beispielweise $44,7\%$ höher als vorher. Bei den Benchmarks ADPCM Decoder, FIR, GSM, ME und SHA kommt es teilweise zu Verschlechterungen, die aber sowohl bei der WCET als auch bei der ACET in einer ähnlichen Höhe auftreten. Die Erhöhung der WCET um $34,2\%$ bei Bubblesort für den TriCore wird, wie bereits bei Optimierungstufe O1, durch zusätzliche Speicherzugriffe verursacht.

Bei GSM sind die Auswirkungen für die verschiedenen Architekturen sehr unterschiedlich. Beim ARM7 mit dem ARM-Instruktionssatz erhöht sich die ACET um $9,9\%$, während die WCET um $25,4\%$ reduziert wird. Beim Thumb-Instruktionssatz kommt es bei WCET und ACET zu Verschlechterungen um $12,6\%$ bzw. $10,9\%$. Beim TriCore reduziert sich die ACET um $4,5\%$ und die WCET steigt um $44,7\%$. Die Unterschiede bei der WCET werden durch die Loop Bound Annotationen für eine Schleife verursacht, deren Iterationsanzahl sich bei verschiedenen Aufrufen stark unterscheidet. Beim ARM7 mit ARM-Instruktionssatz wird die Iterationsanzahl dieser

Schleife vor der Optimierung durch eine Loop Bound Annotation angegeben, und es kommt zu einer hohen WCET-Überschätzung. Nach der Optimierung werden die Loop Bounds von aiT automatisch erkannt, und die Überschätzung wird stark reduziert. Beim Thumb-Instruktionssatz werden die Loop Bounds in keinem Fall automatisch bestimmt, sodass die Verschlechterungen der WCET und der ACET etwa gleich hoch sind. Beim TriCore werden die Loop Bounds vor der Optimierung erkannt, aber nachher nicht mehr, wodurch sich eine Verschlechterung der WCET ergibt, obwohl die ACET verbessert wurde.

Die Unterschiede bei der automatischen Erkennung der Loop Bounds werden hauptsächlich durch die ab Optimierungsstufe O2 durchgeführte Optimierung *Transform Head Controlled Loops* verursacht. Beim TriCore werden nach dieser Optimierung sehr viel weniger Loop Bounds erkannt, und durch die somit nötigen ungenaueren manuellen Angaben wird die WCET-Überschätzung erhöht, was bei fast allen Benchmarks zu einer Verschlechterung der Gesamt-WCET führt. Diese Auswirkungen sind deutlich sichtbar im Diagramm aus Abbildung 6.4, in dem die relative WCET und ACET nach der einzeln durchgeführten Optimierung *Transform Head Controlled Loops* dargestellt ist. Für alle Benchmarks und Architekturen ergeben sich Reduzierungen der ACET zwischen $27,1\%$ (CRC/ARM) und 1% (ADPCM Coder/TriCore). Beim ARM7 wurde auch in allen Fällen die WCET um $35,9\%$ (GSM/ARM) bis $0,1\%$ (GSM/Thumb) reduziert. Durch die verbesserte Loop Bound Analyse sind die Verbesserungen zum Teil höher als die entsprechenden ACET-Verbesserungen. Beim TriCore liegt die maximale WCET-Verbesserung dagegen nur bei $0,9\%$ (ME). Bei sechs der zehn Benchmarks verschlechterte sich die WCET um bis zu $52,9\%$ (GSM).

Die durchschnittliche Reduzierung der WCET über alle Benchmarks beträgt bei der Optimierungsstufe O2 $10,7\%$ (ARM) bzw. $6,9\%$ (Thumb). Beim TriCore ergibt sich im Durchschnitt wieder eine Verschlechterung von $9,4\%$. Die durchschnittliche ACET-Verbesserung beträgt $5,1\%$ (ARM), $5,4\%$ (Thumb) bzw. $8,9\%$ (TriCore).

Bei der Optimierungsstufe O3 ergeben sich nur wenige Änderungen gegenüber der Optimierungsstufe O2. Bei den Benchmarks CRC, G.721, GSM und SHA wurden durch *Loop Unrolling* und *Inline Expansion* weitere Verbesserungen erreicht. Die Reduzierungen der ACET und der WCET liegen dabei jeweils in der gleichen Größenordnung. Bei CRC für den ARM-Instruktionssatz des ARM7 beträgt die WCET-Reduzierung beispielsweise genau wie die ACET-Reduzierung $31,9\%$. Lediglich bei G.721 für den Thumb-Instruktionssatz des ARM7 ist die WCET- und ACET-Reduzierung durch ungünstiges Inlining einer Funktion geringer als bei der Optimierungsstufe O2. Beim G.721 Decoder beträgt die WCET-Verbesserung bei der Optimierungsstufe O3 $8,2\%$ und die ACET-Verbesserung $5,8\%$, im Gegensatz zu einer Verbesserung der WCET um $17,7\%$ und der ACET um $13,2\%$ bei der Optimierungsstufe O2. Ansonsten treten die bei den Ergebnissen der Optimierungstufe O2 beschriebenen Besonderheiten auch weiterhin bei der Optimierungsstufe O3 auf.

Bei der Optimierungsstufe O3 beträgt die durchschnittliche Reduzierung der WC-

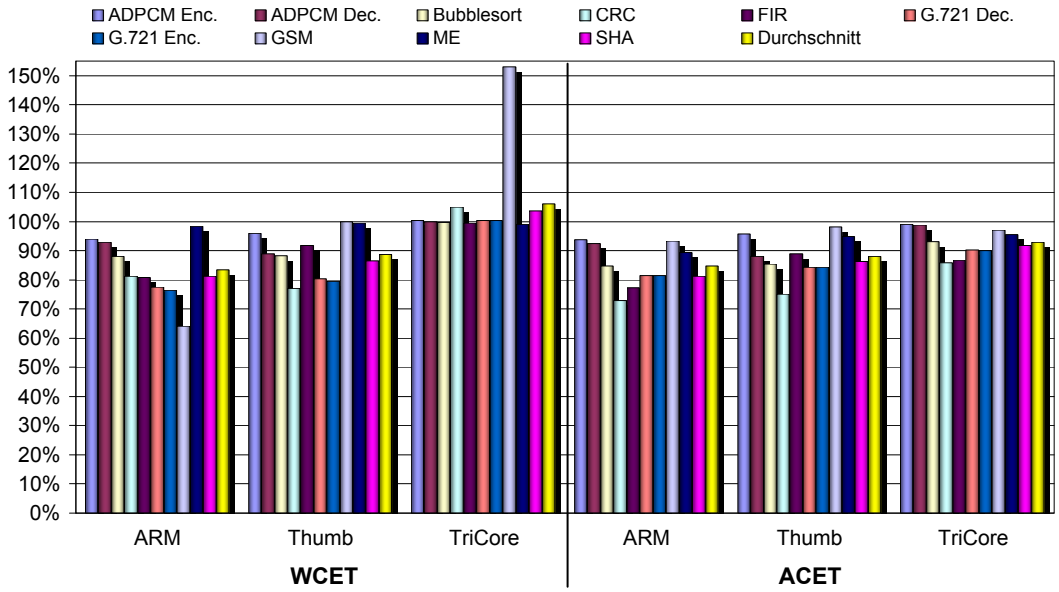

Abbildung 6.4: Relative WCET und ACET nach der ICD-C Optimierung Transform Head Controlled Loops

ET über alle Benchmarks 13, 7% (ARM) bzw. 6, 3% (Thumb). Beim TriCore ergibt sich im Durchschnitt auch hier eine Verschlechterung von 6, 1%. Die durchschnittliche ACET-Verbesserung beträgt 8, 2% (ARM), 5, 4% (Thumb) bzw. 10, 3% (TriCore).

Im Gegensatz zu den hier vorgestellten Ergebnissen der kompletten Optimierungsabfolgen des ICD-C Compiler Framework für die drei Optimierungsstufen sind die Auswirkungen der jeweils einzeln durchgeführten Optimierungen nur sehr gering. Für einen großen Teil der Optimierungen ergeben sich keinerlei Änderungen bei ACET und WCET der Benchmarks. Lediglich bei der Optimierung *Transform Head Controlled Loops* wurden alle Benchmarks modifiziert. Dies zeigt, dass viele der Optimierungen erst in Kombination mit anderen Optimierungen, wie in der ICD-C Optimierungsabfolge, zur einer Verbesserung der WCET oder ACET führen. In Anhang A sind die Diagramme mit den Ergebnissen aller einzelnen Optimierungen abgedruckt.

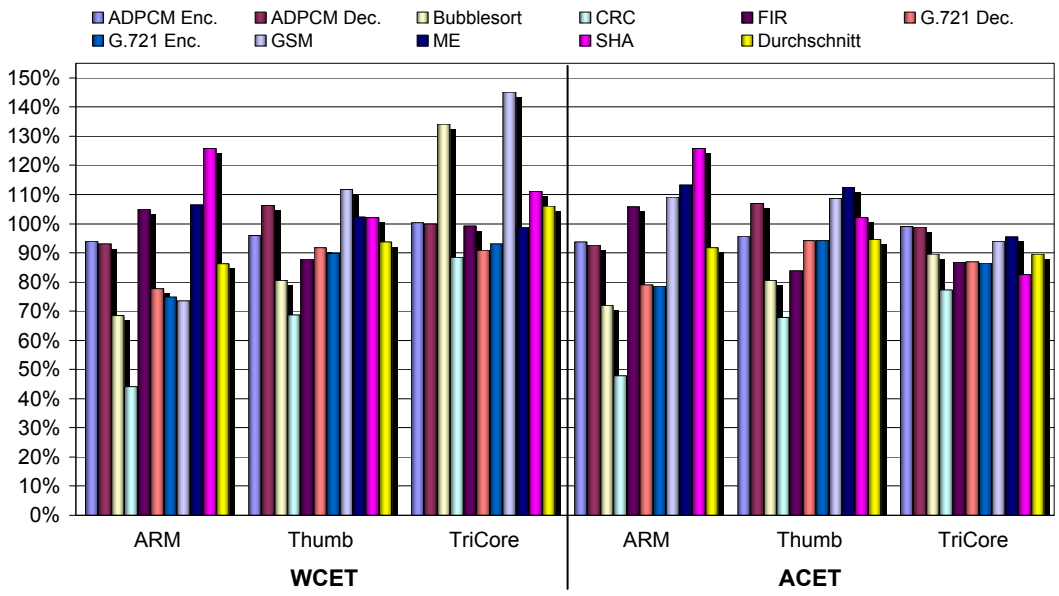

Abbildung 6.5: Relative WCET und ACET nach ICD-C Optimierung O3

7 Zusammenfassung und Ausblick

In diesem letzten Kapitel der vorliegenden Arbeit werden die durchgeführten Untersuchungen sowie die erzielten Ergebnisse noch einmal kurz zusammengefasst, und es wird ein Ausblick gegeben auf mögliche weiterführende Arbeiten.

7.1 Zusammenfassung

Eingebettete Systeme müssen häufig Echtzeit-Anforderungen genügen. Für das Scheduling bei Echtzeitsystemen wird eine obere Laufzeitschranke der auszuführenden Programme benötigt. Mit Hilfe von WCET-Analyse Werkzeugen können solche WCET-Abschätzungen berechnet werden. Da bei solchen Systemen meist auch Faktoren wie Performance, Energieeffizienz und Kosten eine große Rolle spielen, sollte die WCET möglichst gering sein. Heutige Compiler führen meist viele Optimierungen zur Reduzierung der durchschnittlichen Programmlaufzeit durch. Da diese Optimierungen nicht notwendigerweise die gleichen Auswirkungen auf die WCET haben, wäre es wünschenswert, die WCET-Reduzierung als eigenes Optimierungsziel in einen Compiler zu integrieren. Mit den in dieser Arbeit durchgeführten Untersuchungen des Einflusses von Compiler-Optimierungen auf die WCET sollten Optimierungen ermittelt werden, die sich für die Integration in einen solchen WCET-reduzierenden Compiler eignen.

Ein erstes Ziel der Arbeit war es, Faktoren zu identifizieren, die einen Einfluss auf die WCET-Abschätzung haben, und Compiler-Optimierungen nach einer möglichen Beeinflussung dieser Faktoren zu klassifizieren. In Kapitel 3 wurde eine solche Klassifizierung vorgestellt. Es wurden die fünf Einflussfaktoren Loop Bounds, Ergebnisse der Value Analyse, Cache Verhalten, Pipeline Verhalten und Kontrollfluss beschrieben, und diesen insgesamt 16 verschiedene Compiler-Optimierungen zugeordnet. Anhand dieser Klassifizierung wurden vier Optimierungen für eine genauere Untersuchung ausgewählt. Dies waren zum einem die drei Kontrollflussoptimierungen *Loop Nest Splitting*, *Procedure Cloning* und *Loop Unrolling*, und zum anderen das *Code Positioning* zur direkten Optimierung des Worst-Case Pfades.

Beim *Loop Nest Splitting* wird der Iterationsbereich tief verschachtelter Schleifen aufgeteilt, so dass in einem Teil des Iterationsbereiches *if*-Anweisungen in der innersten Schleife entfallen können. Dadurch werden die Instruktionen zur Auswertung der Bedingungen eingespart, und es werden Sprünge vermieden, die *pipeline stalls* zur Folge hätten. Es hat sich gezeigt, dass durch das Loop Nest Splitting zum Teil

erhebliche Verbesserungen der WCET möglich sind, die für einen Benchmark mit bis zu 89% sogar deutlich über den erreichten Verbesserungen der ACET von 59, 4% lagen. Diese WCET-Verbesserungen von durchschnittlich 34% bis 35, 7% wurden erst dadurch erreicht, dass genaue Informationen über die Ausführungsanzahl der verschiedenen Pfade in Form von *flow*-Annotation bei der WCET-Analyse angegeben wurden. Diese Informationen können leicht durch den Optimierungsalgorithmus berechnet werden. Die Verbesserungen der WCET, die über denen der ACET liegen, sind auf eine erhebliche Reduzierung der WCET-Überschätzung zurückzuführen, die durch die Annotationen erreicht wird. Desweiteren wurde eine Möglichkeit zur Abwägung zwischen der Erhöhung der Codegröße und der Reduzierung der WCET beim *Loop Nest Splitting* vorgestellt, die durch eine Platzierung der aufteilenden *if*-Anweisung auf verschiedenen Ebenen der verschachtelten Schleife möglich ist. Die Ergebnisse dieser Untersuchungen wurden bereits in [FS06a] und [FS06b] separat veröffentlicht.

Als zweite Kontrollfluss-Optimierung wurde das *Procedure Cloning* untersucht, bei dem spezialisierte Versionen von Funktionen erzeugt werden, indem einige der Funktionsparameter durch Konstanten ersetzt werden. Während bei der ACET nur Verbesserungen von wenigen Prozent erreicht wurden, die auf den geringeren Aufwand beim Funktionsaufruf und ermöglichte Optimierungen der geklonten Funktionen, wie z.B. *Constant Folding*, zurückzuführen sind, wurden bei der WCET in fast allen Fällen sehr viel höhere Reduzierungen um bis zu 96% erreicht. Diese Verbesserungen ergeben sich im Wesentlichen dadurch, dass die Loop Bound Annotationen nach der Optimierung für die verschiedenen Versionen der geklonten Funktion separat angegeben werden konnten. So wurde die WCET-Überschätzung, verursacht durch konstante Loop Bounds für alle Ausführungen einer Schleife, erheblich reduziert. Da aiT intern unterschiedliche Loop Bounds in Abhängigkeit vom Kontext unterstützt, wäre es bei einer Integration von aiT in eine Compilerumgebung mit direktem Zugriff auf aiT's Zwischendarstellung möglich, die Loop Bounds für verschiedene Aufrufe einer Funktion unterschiedlich anzugeben. Dadurch könnten vermutlich ähnliche Verbesserungen ohne eine Erhöhung der Codegröße erreicht werden, und die WCET-Verbesserungen durch das *Procedure Cloning* würden sehr viel geringer ausfallen.

Die letzte untersuchte Kontrollfluss-Optimierung ist das *Loop Unrolling*. Beim *Loop Unrolling* werden Schleifen abgerollt, indem der Schleifenkörper mehrfach ausgeführt und die Iterationsanzahl entsprechend reduziert wird. Dadurch wird die Anzahl der Sprünge reduziert und somit auch *pipeline stalls* vermieden. Bei einer nicht konstanten Iterationsanzahl müssen zusätzliche Kontrollstrukturen zur Behandlung der letzten Iterationen eingefügt werden. Die Auswahl der zu optimierenden Schleifen erfolgte hier anhand der Informationen einer vorherigen WCET-Analyse. Es wurden Schleifen gewählt, die eine hohe Ausführungsanzahl im ermittelten Worst-Case Szenario und eine hohe maximale Iterationsanzahl haben, wodurch sich ein großer Einfluss auf die Gesamt-WCET ergibt. Bei den Experimenten hat sich ge-

zeigt, dass sich bei Schleifen mit einer konstanten Iterationsanzahl die WCET-Reduzierung sehr ähnlich zur ACET-Reduzierung verhält. Musste allerdings eine zusätzliche Schleife eingefügt werden, so fielen insbesondere bei hohen Unrolling-Faktoren die WCET-Verbesserungen sehr gering aus. Es wurde versucht, durch Angabe von *flow*-Annotationen eine exaktere Berücksichtigung der Iterationsanzahlen zu erreichen. Allerdings konnten damit nur geringe Verbesserungen erreicht werden, während sich der Analyseaufwand deutlich erhöht hat.

In Kapitel 5 wurde eine Optimierung des Worst-Case Pfades mittels *Code Positioning* untersucht. Ziel des *Code Positioning* ist es, die Basisblöcke eines Programms so im Speicher anzuordnen, dass die Anzahl der Sprünge auf einem Ausführungspfad reduziert wird. Bisher wurde dabei meist der am häufigsten ausgeführte Pfad optimiert. Hier wurde stattdessen der Worst-Case Pfad optimiert, indem die Ergebnisse einer vorangegangenen WCET-Analyse bei der Durchführung der Optimierung verwendet wurden. Dabei musste auch berücksichtigt werden, dass sich der WC Pfad durch das Verschieben einzelner Blöcke leicht ändern kann. Obwohl jeweils nur wenige Basisblöcke verschoben wurden, konnten WCET-Reduzierungen von bis zu $8,4\%$ erreicht werden. Die ACET-Reduzierung waren in allen Fällen geringer als die entsprechende WCET-Reduzierung. Teilweise gab es auch Verschlechterungen bei der ACET, was die Unterschiede zwischen dem WC Pfad und dem Pfad einer üblichen Ausführung verdeutlicht.

Mit diesen Untersuchungen konnten somit die Ziele der Arbeit erreicht werden. Es sollten Compiler-Optimierungen gefunden werden, die sich gut für die Integration in einen WCET-optimierenden Compiler eignen. Außerdem sollten Möglichkeiten zur Nutzung von WCET-Informationen bei der Optimierung oder Weitergabe von Informationen, die während der Optimierungen gewonnen werden, an die WCET-Analyse untersucht werden. Beim *Loop Nest Splitting*, *Procedure Cloning* und *Code Positioning* wurden WCET-Verbesserungen erreicht, die über den ACET-Verbesserungen liegen. Beim *Loop Nest Splitting* wurden durch die Optimierung Annotationen für die WCET-Analyse generiert, und beim *Loop Unrolling* sowie beim *Code Positioning* wurden WCET-Information bei der Durchführung der Optimierung verwendet.

Im letzten Teil dieser Arbeit wurden schließlich die Auswirkungen automatisch durchgeführter Standard Optimierungen eines Compilers auf die WCET untersucht. Dazu wurden die 21 Optimierungen des ICD-C Compiler Frontends sowohl einzeln als auch in kompletten Optimierungsabfolgen auf einer Vielzahl verschiedener Benchmarks durchgeführt. Es hat sich gezeigt, dass die Änderungen bei der WCET und der ACET (sowohl Verbesserungen als auch Verschlechterungen) in den meisten Fällen sehr ähnlich waren. Die wenigen Ausnahmen waren meist auf die spezielle Struktur einer Funktion oder Besonderheiten bei der Codegenerierung für bestimmte Architekturen zurückzuführen.

7.2 Ausblick

In dieser Arbeit wurden die in den Kapiteln 4 und 5 untersuchten Optimierungen manuell auf dem Quelltext ausgewählter Benchmarks durchgeführt. Als nächster Schritt könnten nun die Optimierungen, mit denen hier gute Resultate bei der WCET-Reduzierung erzielt wurden, in einem WCET-optimierenden Compiler implementiert werden. Am Lehrstuhl Informatik XII der Universität Dortmund wird derzeit eine Compiler-Umgebung namens *WCC* entwickelt, in die das WCET-Analyse Werkzeug aiT integriert ist [FLT06a, FLT06b]. Damit wird ein Informationsaustausch zwischen Compiler und WCET-Analyse möglich. Bei der Implementierung von Compiler-Optimierungen kann somit auf Ergebnisse der WCET-Analyse zurückgegriffen werden, wie es in dieser Arbeit manuell durchgeführt wurde. Auch das Erzeugen zusätzlicher Annotation während einer Optimierung, wie beispielsweise beim *Loop Nest Splitting* in Kapitel 4.1.2, kann leicht implementiert werden.

Die *WCC* Compiler-Umgebung basiert auf dem ICD-C Compiler Frontend, welches auch für die Untersuchungen in Kapitel 6 verwendet wurde. Die ICD-C Zwischendarstellung wird zur Codeerzeugung an ein eigenes Compiler-Backend für die TriCore v1.3 Architektur weitergegeben. Erste Tests mit diesem Compiler haben gezeigt, dass sich die Auswirkungen der ICD-C Optimierungen auf die WCET erheblich von denen der Untersuchung in Kapitel 6 unterscheiden. In einer genaueren Untersuchung könnten die Unterschiede bei der Codeerzeugung analysiert werden, die zu den unterschiedlichen Ergebnissen führen.

A Ergebnisse der ICD-C Optimierungen

Die in diesem Anhang abgedruckten Diagramme stellen die relative WCET und ACET nach jeweils einer einzeln durchgeführten Optimierung des ICD-C Compiler Frameworks dar (siehe Kapitel 6).

Es werden die prozentuale WCET und ACET nach der jeweiligen Optimierung angegeben, relativ zur den Werten der unoptimierten Versionen, welche 100% entsprechen.

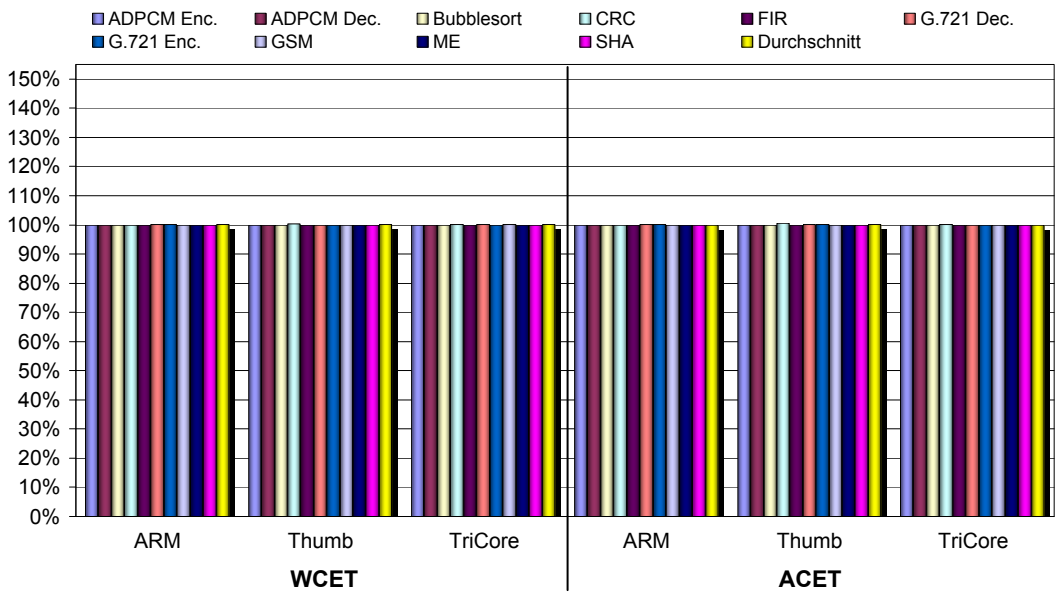

Abbildung A.1: ICD-C Optimierung Create Multiple Exits

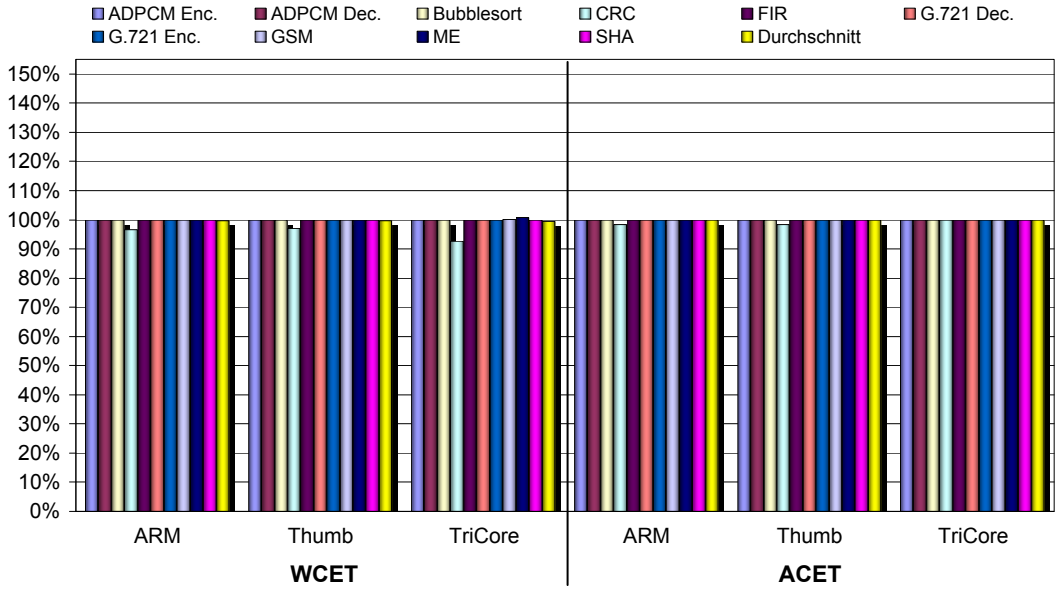

Abbildung A.2: ICD-C Optimierung Dead Code Elimination

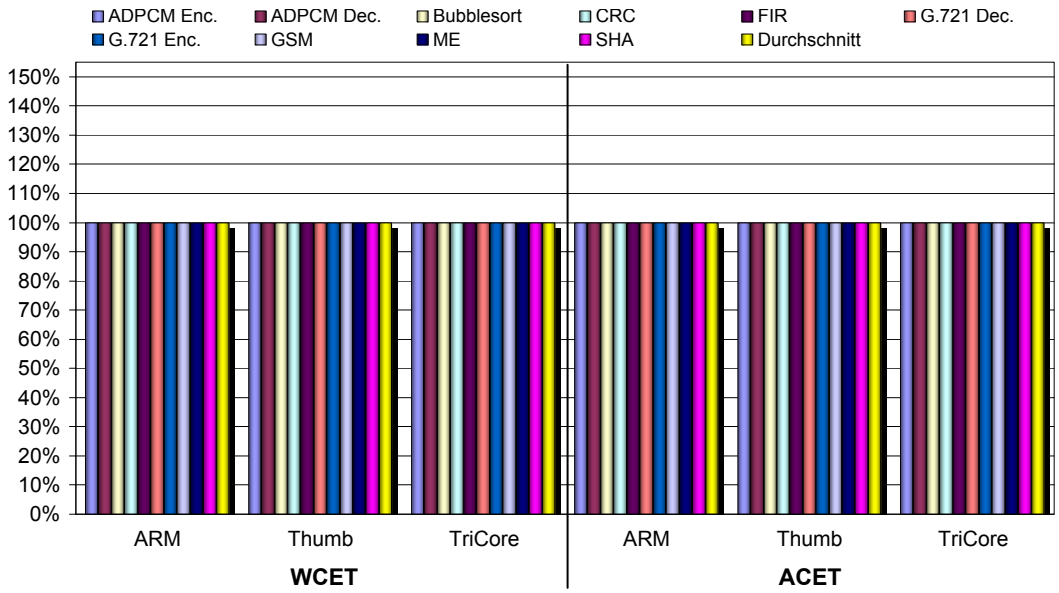

Abbildung A.3: ICD-C Optimierung Eliminate Return Value

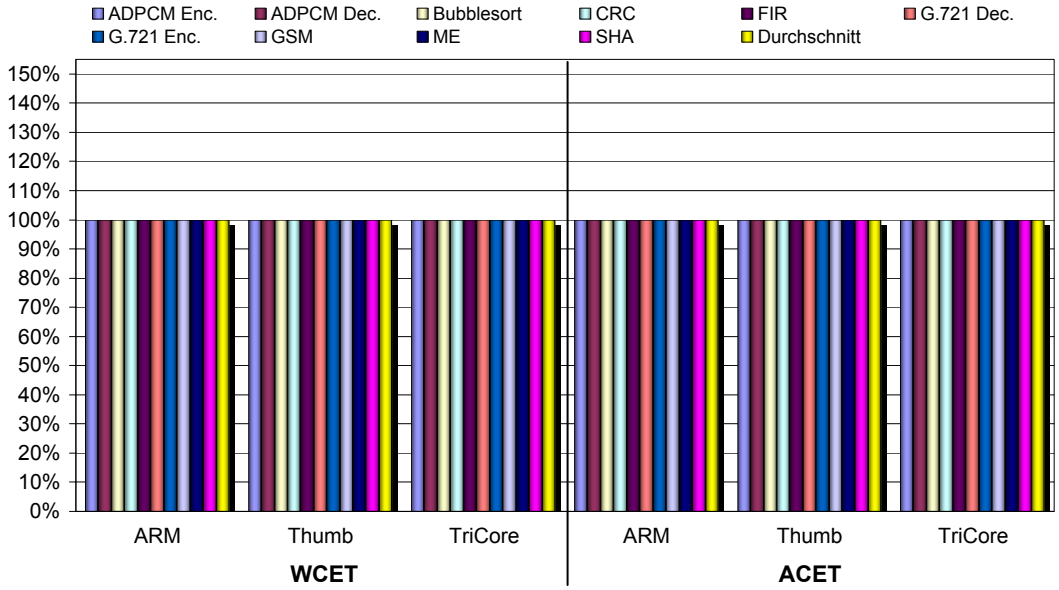

Abbildung A.4: ICD-C Optimierung Eliminate Tail Recursion

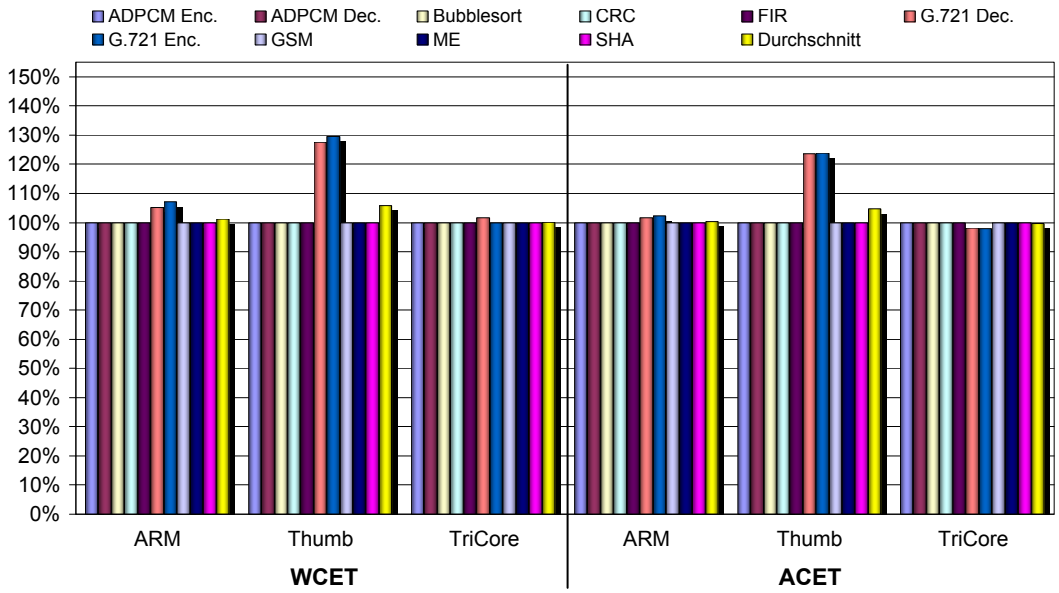

Abbildung A.5: ICD-C Optimierung Inline Expansion

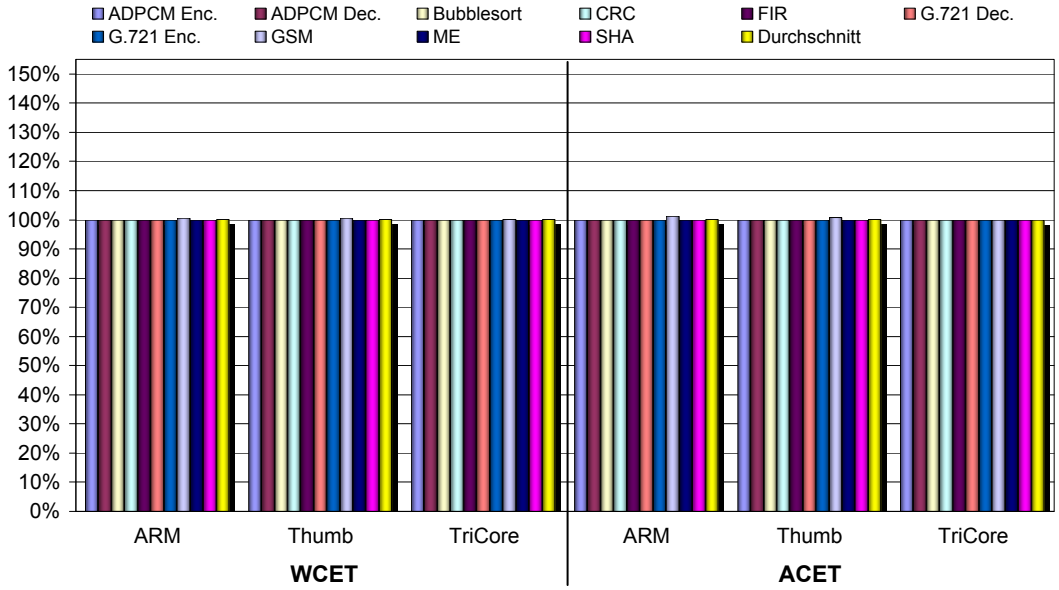

Abbildung A.6: ICD-C Optimierung Fold Constant Code

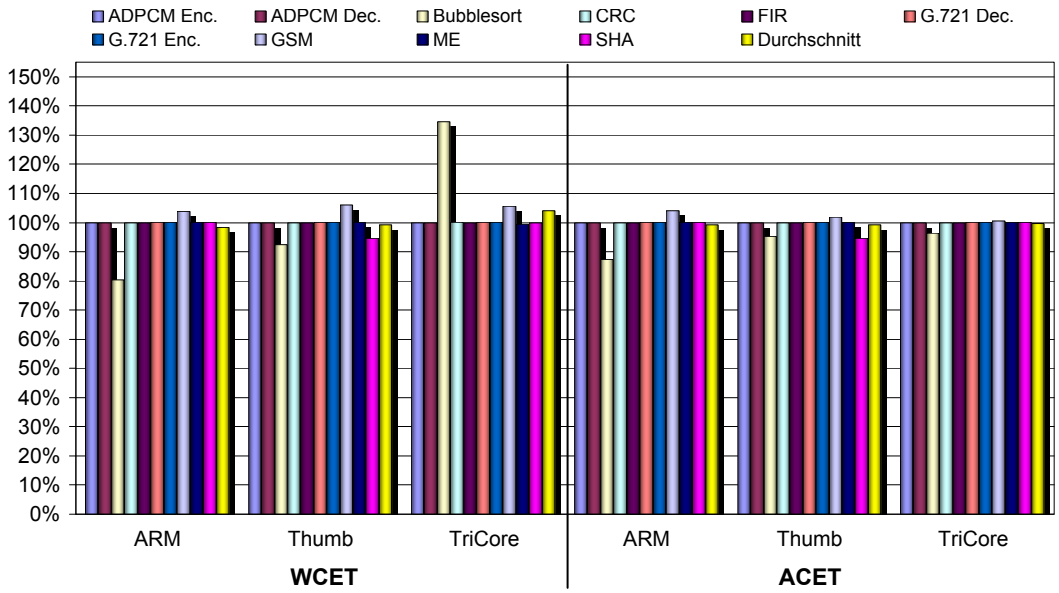

Abbildung A.7: ICD-C Optimierung Common Subexpression Elimination

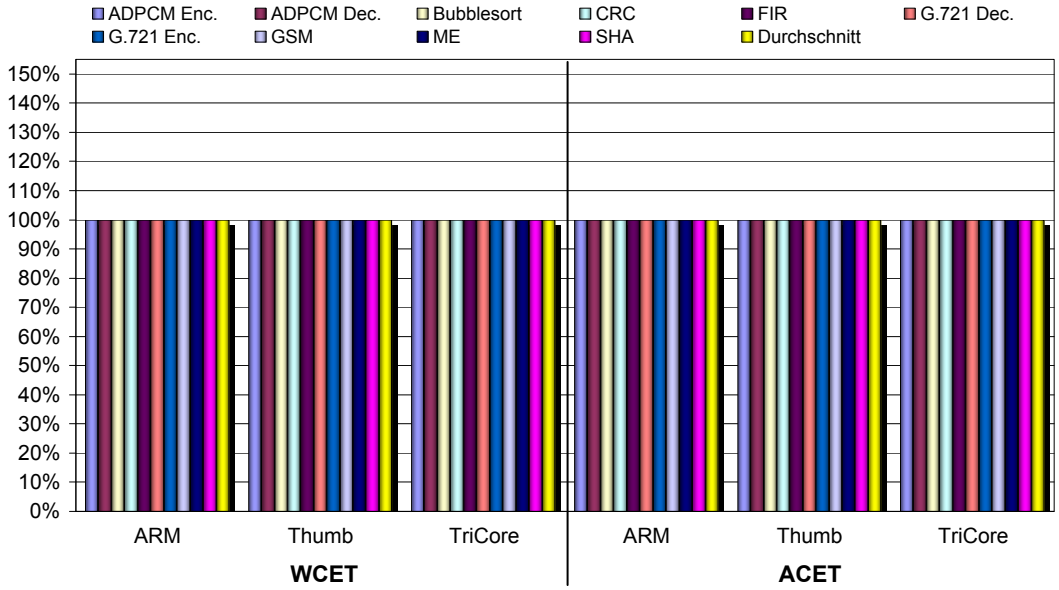

Abbildung A.8: ICD-C Optimierung Loop Collapsing

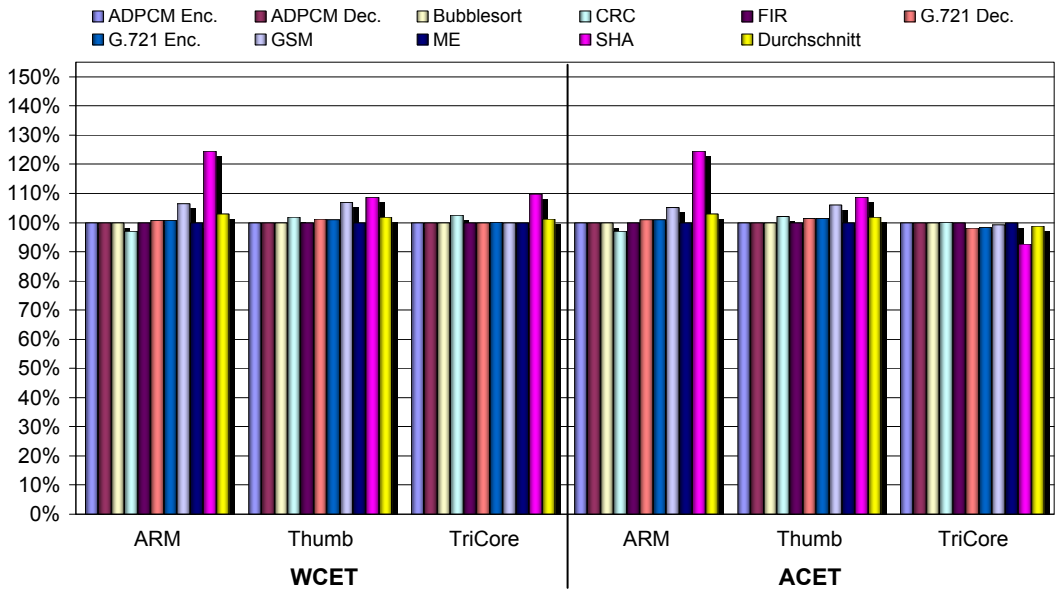

Abbildung A.9: ICD-C Optimierung Loop De-Indexing

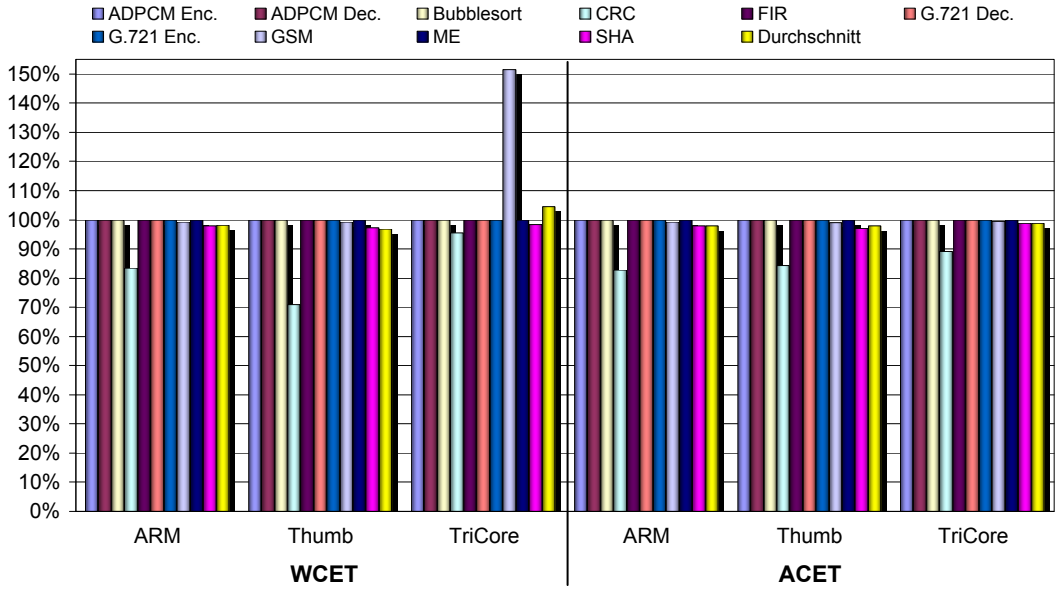

Abbildung A.10: ICD-C Optimierung Loop Unrolling

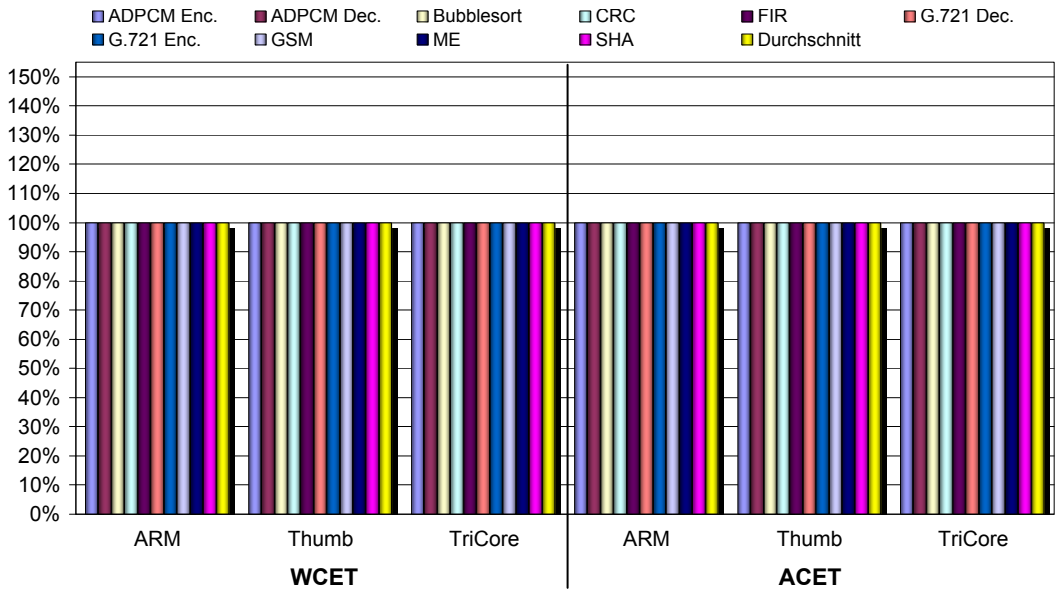

Abbildung A.11: ICD-C Optimierung Loop Unswitching

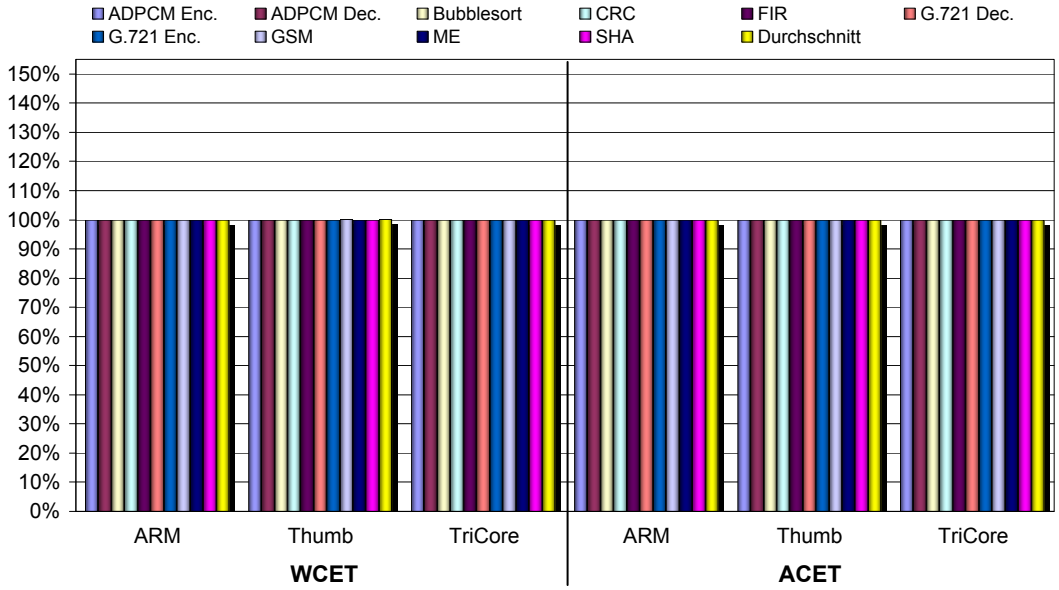

Abbildung A.12: ICD-C Optimierung Merge String Constant Expressions

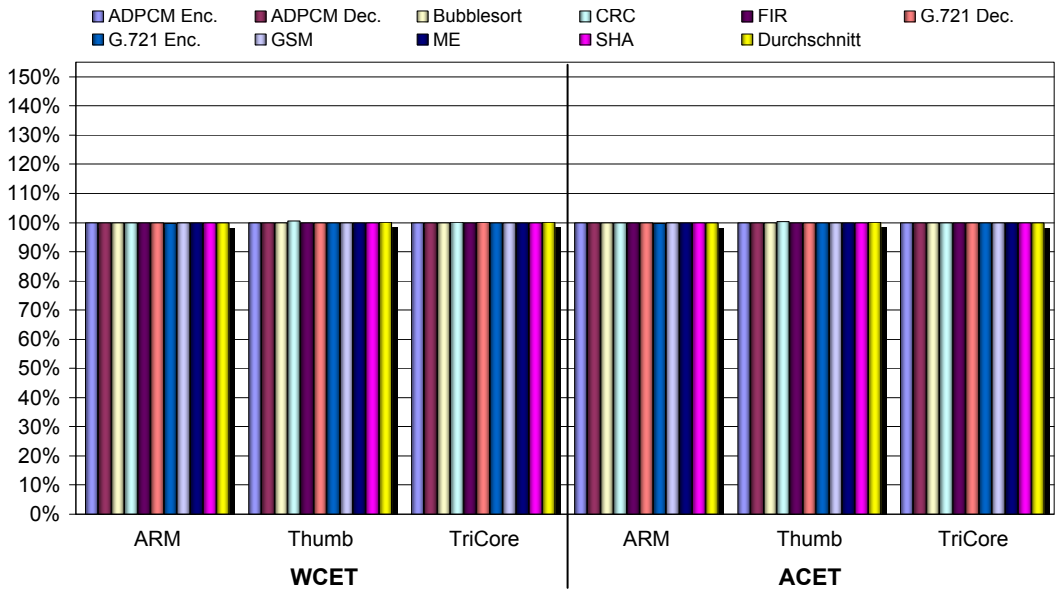

Abbildung A.13: ICD-C Optimierung Value Propagation

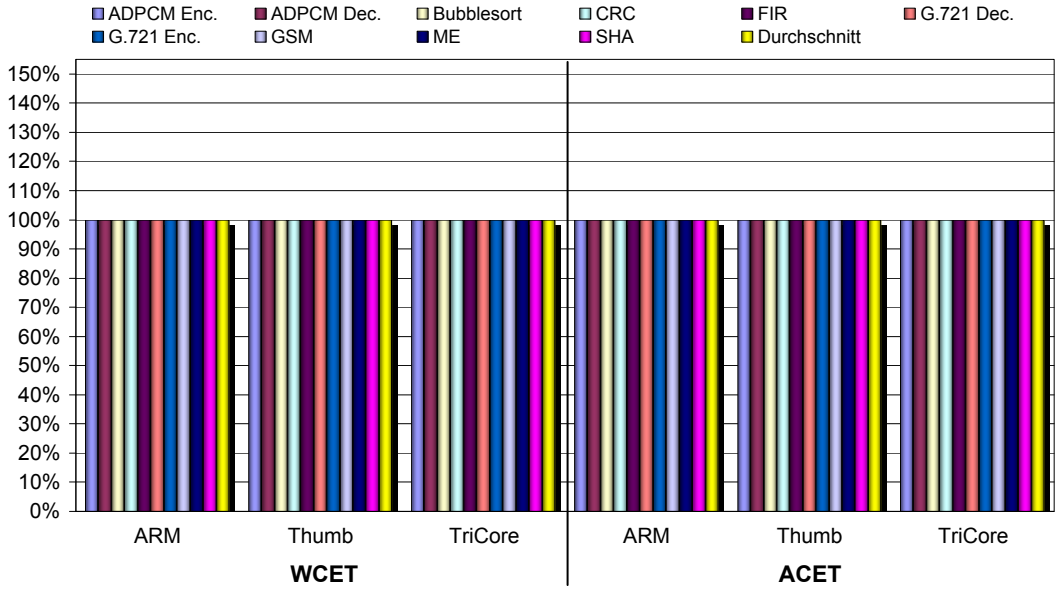

Abbildung A.14: ICD-C Optimierung Redundant Load Elimination

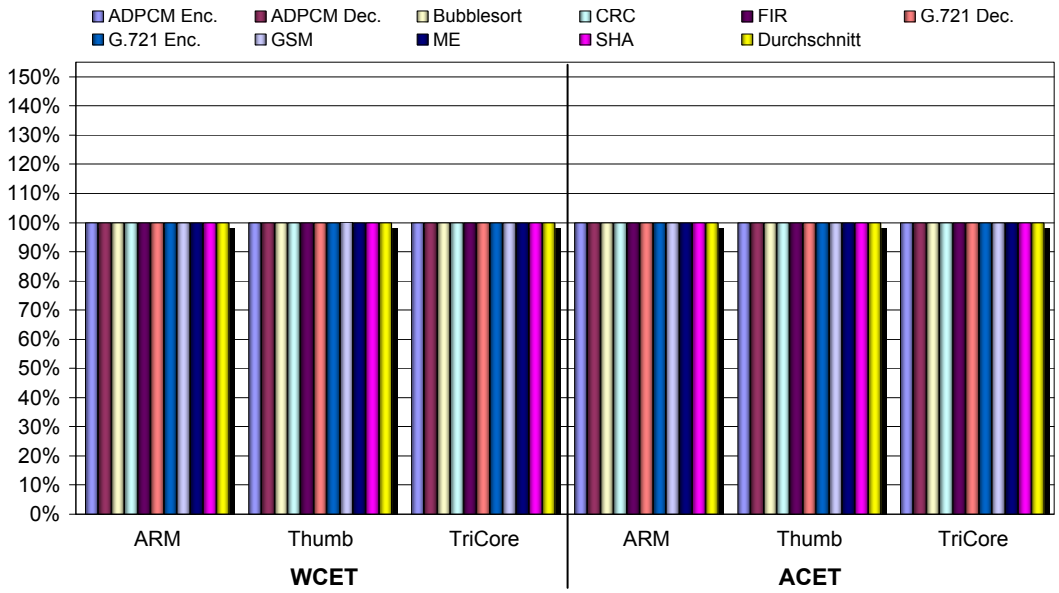

Abbildung A.15: ICD-C Optimierung Remove Unused Function Arguments

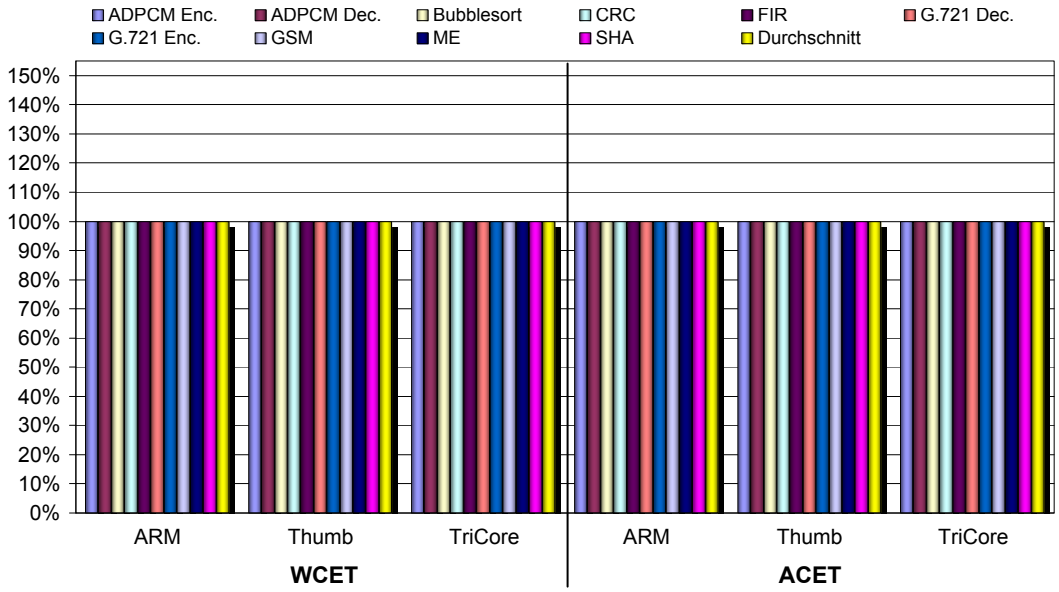

Abbildung A.16: ICD-C Optimierung Remove Unused Symbols (lokal)

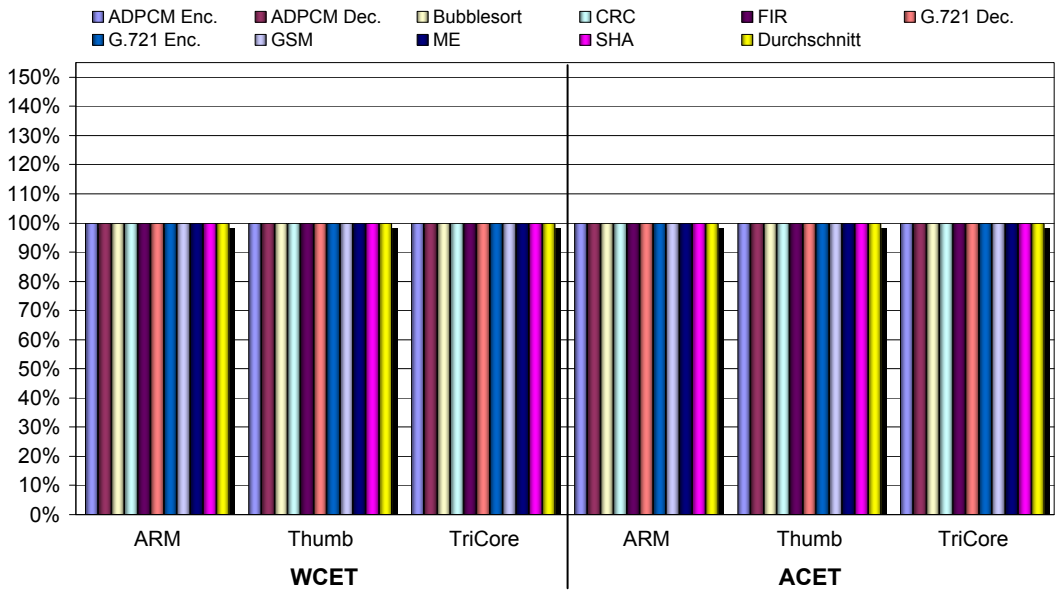

Abbildung A.17: ICD-C Optimierung Remove Unused Symbols

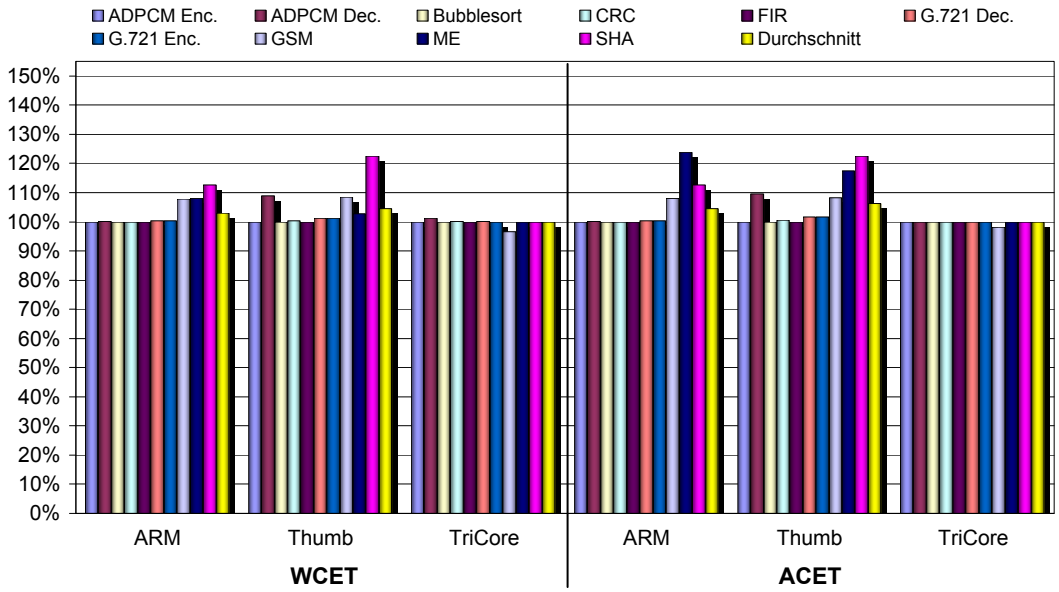

Abbildung A.18: ICD-C Optimierung Separate Life Ranges

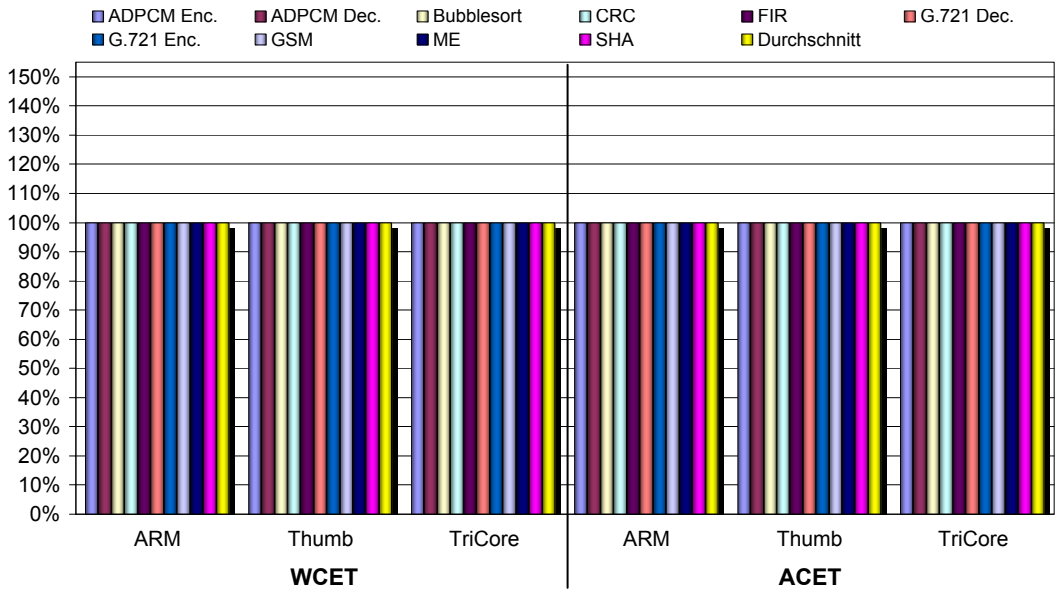

Abbildung A.19: ICD-C Optimierung Expression Simplification

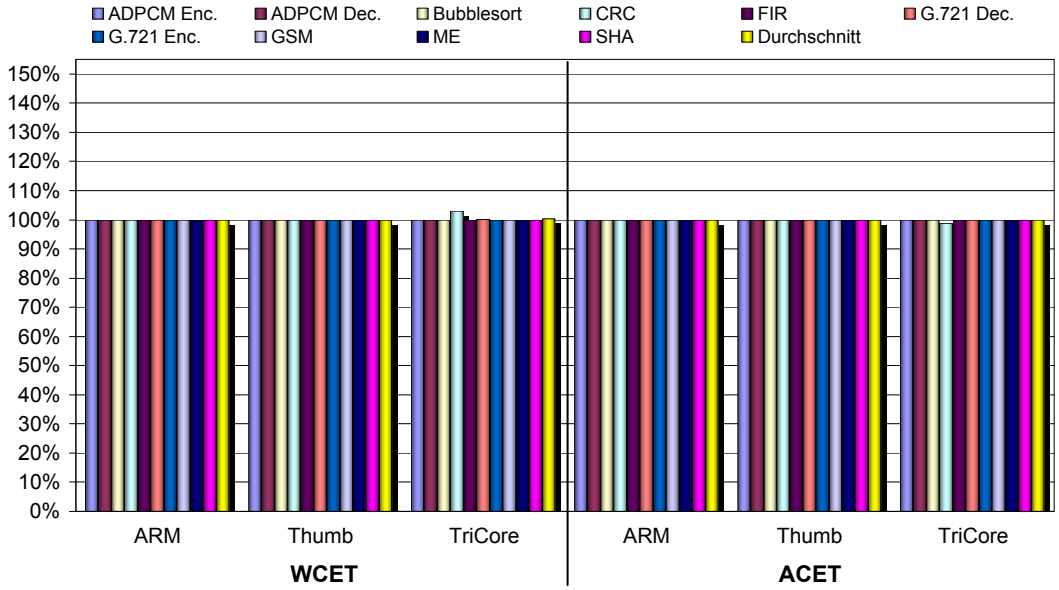

Abbildung A.20: ICD-C Optimierung Function Specialization

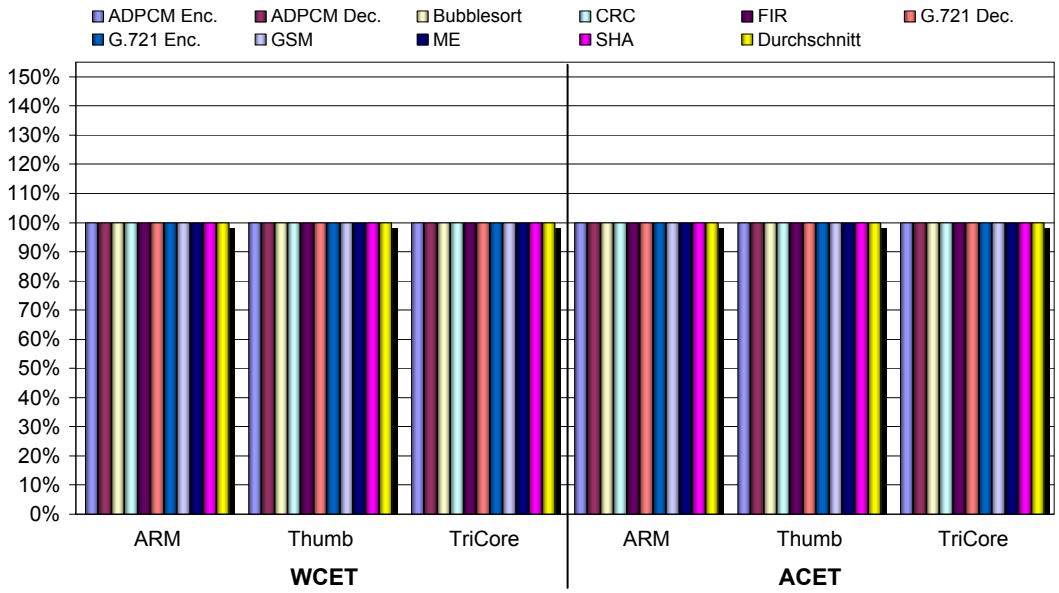

Abbildung A.21: ICD-C Optimierung Struct Scalarization

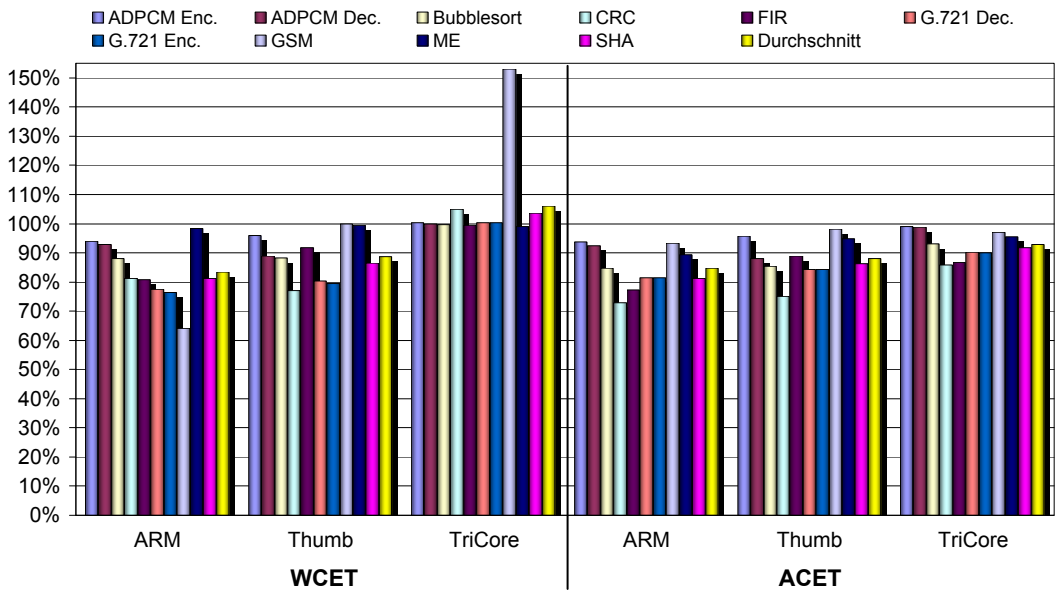

Abbildung A.22: ICD-C Optimierung Transform Head Controlled Loops

Literaturverzeichnis

[Abs06a] AbsInt Angewandte Informatik GmbH: *aiT Worst-Case Execution Time Analyzers*. `http://www.absint.com/ait/`. Version: 2006

[Abs06b] AbsInt Angewandte Informatik GmbH (Hrsg.): *Worst-Case Execution Time Analyzer aiT for ARM7*. Saarbrücken: AbsInt Angewandte Informatik GmbH, 2006

[Abs06c] AbsInt Angewandte Informatik GmbH (Hrsg.): *Worst-Case Execution Time and Stack Analysis aiT for TriCore*. Saarbrücken: AbsInt Angewandte Informatik GmbH, 2006

[ARM98] ARM Limited (Hrsg.): *ARM Software Development Toolkit User Guide*. Version 2.50. Cambridge, UK: ARM Limited, 1998

[ARM01] ARM Limited (Hrsg.): *ARM7TDMI-S (Rev 4) Technical Reference Manual*. Cambridge, UK: ARM Limited, 2001

[ASU86] Aho, A. V. ; Sethi, R. ; Ullman, J. D.: *Compilers: Principles, Techniques, and Tools*. Reading, Mass. : Addison-Wesley, 1986. – ISBN 0–201–10194–7

[BGS94] Bacon, David F. ; Graham, Susan L. ; Sharp, Oliver J.: Compiler Transformations for High-Performance Computing. In: *ACM Computing Surveys* 26 (1994), Nr. 4, S. 345–420. – ISSN 0360–0300

[BH03] Bernat, G. ; Holsti, N.: Compiler Support for WCET Analysis: a Wish List. In: *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*, 2003

[BTC89] Bister, M. ; Taeymans, Y. ; Cornelis, J.: Automatic segmentation of cardiac MR images. In: *IEEE Journal on Computers in Cardiology* (1989), S. 215–218

[CC77] Cousot, P. ; Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. Los Angeles, Jan. 1977

[EES+99] Engblom, J. ; Ermedahl, A. ; Sjödin, M. ; Gustafsson, J. ; Hansson, H.: Towards industry-strength worst case execution time analysis / Advanced Software Technology Center. 1999 (ASTEC 99/02). – Forschungsbericht

[FLT06a] Falk, Heiko ; Lokuciejewski, Paul ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET 2006).* Dresden, Jul. 2006

[FLT06b] Falk, Heiko ; Lokuciejewski, Paul ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: *Proceedings of the 4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2006).* Seoul, Okt. 2006

[FM03] Falk, Heiko ; Marwedel, Peter: Control Flow driven Splitting of Loop Nests at the Source Code Level. In: *Proceedings of the conference on Design, Automation and Test in Europe (DATE '03)*, 2003

[FM04] Falk, Heiko ; Marwedel, Peter: *Source Code Optimization Techniques for Data Flow Dominated Embedded Software.* Dordrecht : Kluwer Academic Publishers, 2004. – ISBN 1–4020–2822–9

[FMWA99] Ferdinand, Christian ; Martin, Florian ; Wilhelm, Reinhard ; Alt, Martin: Cache behavior prediction by abstract interpretation. In: *Science of Computer Programming* 35 (1999), Nr. 2-3, S. 163–189. – ISSN 0167–6423

[FS06a] Falk, Heiko ; Schwarzer, Martin: Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET 2006).* Dresden, Jul. 2006

[FS06b] Falk, Heiko ; Schwarzer, Martin: Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In: *Proceedings of the 4th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia 2006).* Seoul, Okt. 2006

[GMCG00] Gupta, Sumit ; Miranda, Miguel ; Catthoor, Francky ; Gupta, Rajesh: Analysis of high-level address code transformations for programmable processors. In: *Proceedings of the conference on Design, automation and test in Europe (DATE '00).* Paris, Mär. 2000

[GRE+01] Guthaus, M. ; Ringenberg, J. ; Ernst, D. ; Austin, T. ; Mudge, T. ; Brown, T.: MiBench: A free, commercially representative embedded

benchmark suite. In: *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characteristics (WWC-4)*. Austin, Dez. 2001

[HF06] Heckmann, Reinhold ; Ferdinand, Christian: *Worst Case Execution Time Prediction by Static Program Analysis*. 2006

[Hig06] HighTec EDV-Systeme GmbH: *HighTec GNU C/C++-Compiler*. `http://www.hightec-rt.com`. Version: 2006

[HP96] Hennessy, John L. ; Patterson, David A.: *Computer Architecture: A Quantitative Approach*. 2. Auflage. San Francisco : Morgan Kaufmann, 1996. – ISBN 1–55860–329–8

[ICD06] ICD – Informatik Centrum Dortmund: *ICD-C Compiler Framework*. `http://www.icd.de/es/icd-c/`. Version: 2006

[Inf05a] Infineon Technologies AG (Hrsg.): *TC1796 32-Bit Single-Chip Microcontroller*. V1.0. München: Infineon Technologies AG, Jun. 2005

[Inf05b] Infineon Technologies AG (Hrsg.): *TriCore 1 32-bit Unified Processor Core*. V1.3.5. München: Infineon Technologies AG, Feb. 2005

[Int94] Intel (Hrsg.): *MCS 51 Microcontroller Family User's Manual*. Santa Clara: Intel, 1994

[KP05] Kirner, Raimund ; Puschner, Peter: Classification of WCET Analysis Techniques. In: *Proceedings of the 8th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC '05)*. Seattle, Mai 2005

[LBJ+95] Lim, Sung-Soo ; Bae, Young H. ; Jang, Gyu T. ; Rhee, Byung-Do ; Min, Sang L. ; Park, Chang Y. ; Shin, Heonshik ; Park, Kunsoo ; Moon, Soo-Mook ; Kim, Chong S.: An Accurate Worst Case Timing Analysis for RISC Processors. In: *IEEE Transactions on Software Engineering* 21 (1995), Nr. 7, S. 593–604. – ISSN 0098–5589

[LBSL96] *Kapitel* 8.1. In: Lapsley, Phil ; Bier, Jeff ; Shoham, Amit ; Lee, Edward A.: *DSP Processor Fundamentals: Architectures and Features*. Wiley-IEEE Press, 1996. – ISBN 0–7803–3405–1, S. 91–94

[LLPM03] Lee, Sheayun ; Lee, Jaejin ; Park, Chang Y. ; Min, Sang L.: A Flexible Tradeoff between Code Size and WCET Employing Dual Instruction Set Processors. In: *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis (WCET 2003)*. Porto, Jul. 2003

[LM95] Li, Yau-Tsun S. ; Malik, Sharad: Performance analysis of embedded software using implicit path enumeration. In: *Proceedings of the ACM SIGPLAN 1995 Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS '95)*. La Jolla, Jun. 1995

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

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

[lps06] *lp_solve Homepage*. `http://lpsolve.sourceforge.net`. Version: 2006

[Mar06] Marwedel, Peter: *Embedded System Design*. 2. Auflage. Dordrecht : Springer, 2006. – ISBN 0–387–29237–3

[Muc97] Muchnick, Steven S.: *Advanced Compiler Design and Implementation*. San Francisco : Morgan Kaufmann Publishers, 1997. – ISBN 1–55860–320–4

[Mär06] *Mälardalen WCET research group benchmarks*. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, 2006

[NRM04] Negi, Hemendra S. ; Roychoudhury, Abhik ; Mitra, Tulika: Simplifying WCET Analysis By Code Transformations. In: *Proceedings of the 4th International Workshop on Worst-Case Execution Time Analysis (WCET 2004)*. Catania, Jun. 2004

[OS97] Ottosson, Greger ; Sjödin, Mikael: Worst-Case Execution Time Analysis for Modern Hardware Architectures. In: *Proceedings of the ACM SIGPLAN 1997 Workshop on Languages, Compilers and Tools for Real-Time Systems (LCT-RTS '97)*, 1997

[Pus02] Puschner, Peter: Transforming Execution-Time Boundable Code into Temporally Predictable Code. In: *Proceedings of the IFIP 17th World Computer Congress - TC10 Stream on Distributed and Parallel Embedded Systems (DIPES '02)*. Montréal, Aug. 2002

[RWT+06] Reineke, Jan ; Wachter, Björn ; Thesing, Stefan ; Wilhelm, Reinhard ; Polian, Ilia ; Eisinger, Jochen ; Becker, Bernd: A Definition and Classification of Timing Anomalies. In: *Proceedings of the 6th International Workshop on Worst-Case Execution Time Analysis (WCET 2006)*. Dresden, Jul. 2006

[SF99] Schneider, Jörn ; Ferdinand, Christian: Pipeline behavior prediction for superscalar processors by abstract interpretation. In: *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES '99)*. Atlanta, Mai 1999

[Str88] Strobach, P.: A new technique in scene adaptive coding. In: *Proceedings of the 4th European Signal Processing Conference (EUSIPCO-88)*. Grenoble, Sep. 1988

[SW02] Steinke, Stefan ; Wehmeyer, Lars: *The encc Energy Aware C Compiler Homepage*. `http://ls12-www.cs.uni-dortmund.de/research/encc/`, 2002

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

[ZKW+04] Zhao, Wankang ; Kulkarni, Prasad ; Whalley, David ; Healy, Christopher ; Mueller, Frank ; Uh, Gang-Ryung: Tuning the WCET of Embedded Applications. In: *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'04)*. Toronto, Mai 2004

[ZKW+05] Zhao, Wankang ; Kreahling, William ; Whalley, David ; Healy, Christopher ; Mueller, Frank: Improving WCET by Optimizing Worst-Case Paths. In: *Proceedings of the 11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS '05)*. San Francisco, Mär. 2005

[ZWHM04] Zhao, Wankang ; Whalley, David ; Healy, Christopher ; Mueller, Frank: WCET Code Positioning. In: *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS '04)*. Lissabon, Dez. 2004

Index