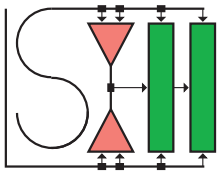
technische universität
dortmund

Diplomarbeit

Transformation und Ausnutzung von WCET-Informationen für High-Level-Optimierungen

Fatih Gedikli



Technische Universität Dortmund
Lehrstuhl Informatik XII

29. September 2008

Gutachter:

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

An dieser Stelle möchte ich mich gerne bei allen Personen bedanken, die zum Gelingen dieser sehr abwechslungsreichen Diplomarbeit beigetragen haben: Allen voran danke ich meinem Betreuer Paul Lokuciejewski für seine großartige Unterstützung in guten wie auch in schwierigen Zeiten. Er hat sich immer Zeit für mich genommen, wenn ich Fragen hatte, und mich bei meinen Ideen unterstützt. Herrn Prof. Dr. Peter Marwedel danke ich dafür, dass er sich als Gutachter zur Verfügung gestellt hat. Außerdem bedanke ich mich bei allen anderen Mitarbeitern des Lehrstuhls 12 für Eingebettete Systeme, insbesondere Sascha Plazar für seine freundliche Unterstützung. Ich danke all denjenigen, die diese Arbeit in kurzer Zeit Korrektur gelesen haben.
Zum Schluss möchte ich meiner Familie danken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACEP	Average-Case Execution Path
ACET	Average-Case Execution Time
BA	Back-annotation
BCET	Best-Case Execution Time
CFG	Control Flow Graph
CISC	Complex Instruction Set Computer
CP	Cold Path
CRL2	Control Flow Representation Language
CSV	Comma Separated Values
DSP	Digital Signal Processor
HLIR	High-Level Intermediate Representation
ICD	Informatik Centrum Dortmund
ILP	Integer Linear Programming
IPET	Implicit Path Enumeration Technique
IR	Intermediate Representation
LLIR	Low-Level Intermediate Representation
LOOCV	Leave-One-Out Cross-Validation
LTA	Life Time Analysis
MAC	Multiply-Accumulate
ML	Machine Learning
MLIR	Medium-Level Intermediate Representation
NN	Near Neighbor
NPU	Network Processing Unit
RISC	Reduced Instruction Set Computer
RPA	Register Pressure Analyzer
SVM	Support Vector Machines
VLIW	Very Long Instruction Word
WCC	WCET-Aware C-Compiler
WCEC$_{sum}$	Summarized Worst-Case Execution Count
WCEP	Worst-Case Execution Path
WCET	Worst-Case Execution Time
WCET$_{est}$	Estimated Worst-Case Execution Time
WCET$_{real}$	Real Worst-Case Execution Time
YALE	Yet Another Learning Environment

1 Einführung

Dieses Kapitel dient als Einführung in die Diplomarbeit. Abschnitt 1.1 motiviert das Themengebiet der Diplomarbeit. Abschnitt 1.2 beschreibt die beiden Ziele der Diplomarbeit. Abschnitt 1.3 präsentiert verwandte Arbeiten. Abschnitt 1.4 stellt schließlich den Aufbau der Diplomarbeit vor.

1.1 Motivation

In immer mehr Produkten sind Hardware- und Software-Komponenten integriert, die für den Benutzer meist nicht direkt sichtbar sind. Dabei handelt es sich um Eingebettete Systeme. Eingebettete Systeme sind „informationsverarbeitende Systeme, die in ein umgebendes Produkt wie z.B. Autos, Telekommunikationsgeräte oder Produktionsmaschinen eingebettet sind“ [Marwedel 2007]. Solche Systeme gewinnen immer mehr an Bedeutung.

„Ob in der Medizintechnik, bei Herzschrittmachern und Magnetresonanztomographen oder der Automation von industriellen Anlagen, überall heißt es: Embedded Systems inside“. Das betonte Bitkom-Vizepräsident Heinz-Paul Bonn in einem Pressegespräch auf der traditionsgemäß für Maschinen und Anlagen bekannten Hannover Messe 2008. So sind in einem aktuellen Mittelklassewagen mittlerweile über 70 Prozessoren integriert. In einem einzigen Auto laufen bereits Software-Komponenten mit 10 Millionen Code-Zeilen und 2015 sollen es sogar 100 Millionen sein [Heise Online News 2008]. Eingebettete Systeme sind zu einem zentralen Bestandteil aktueller Produkte der Industrie geworden, obgleich die Informationsverarbeitung Eingebetteter Systeme nicht ausschlaggebend für den Kauf des übergeordneten Produktes ist. Statt dessen spielt die Nutzung des übergeordneten Produktes eine wichtige Rolle bei der Kaufentscheidung.

Eingebettete Systeme kommen heute in vielen Anwendungsbereichen zum Einsatz. Sie unterstützen den modernen Menschen in vielen Bereichen des täglichen Lebens. Beispiele für wichtige Anwendungsbereiche sind:

- Konsumelektronik
- Multimedia
- Telekommunikation
- Transportmittel
- Gebäude-Automation

- Robotik
- Medizintechnik

Die Anwendungsbereiche stellen besondere Anforderungen an Eingebettete Systeme. Derartige Systeme müssen vor allem effizient sein. Ein Eingebettetes System muss beispielsweise effizient im Hinblick auf Laufzeit, Energieverbrauch, Code-Größe, physikalische Größe, physikalisches Gewicht und Kosten sein. Weitere wichtige Anforderungen sind Sicherheit und Zuverlässigkeit.

Häufig ist ein Eingebettetes System auch ein Realzeit-System, das Zeitschranken bei der Programmausführung einhalten muss. Ein Realzeit-Sytem heißt korrekt, wenn ein korrektes Ergebnis innerhalb der von außen vorgegebenen Zeit berechnet werden kann. Es gibt harte und weiche Realzeit-Systeme. Wenn in einem harten Realzeit-System die von außen vorgegebene Zeit nicht eingehalten werden kann, dann ist das oft gleichbedeutend mit einer Katastrophe. Eine katastrophale Folge ist z.B. der Verlust menschlichen Lebens. So könnte durch eine Fehlfunktion in der Steuerung eines Transportmittels, wie z.B. eines Flugzeugs, ein Unfall mit schwerwiegenden Folgen verursacht werden.

Bei weichen Realzeit-Systemen hingegen gibt es keine katastrophalen Folgen, wenn die von außen vorgegebene Zeit nicht eingehalten werden kann. Meistens haben Multimedia-Systeme aus Kostengründen weiche Realzeit-Anforderungen. Wenn Audio- oder Videodaten in einem Multimedia-System mit weichen Realzeit-Anforderungen nicht rechtzeitig zur Verfügung stehen, dann führt das lediglich zu einer Verschlechterung der Qualität, die aber rufschädigend für den Hersteller sein kann.

Das Einhalten von Zeitschranken ist also eine wichtige Anforderung an Eingebettete Systeme, die zugleich Realzeit-Systeme sind. Um das Einhalten der von außen vorgegebenen Zeit garantieren zu können, muss die maximale Programm-Laufzeit (Worst-Case Execution Time, WCET) bekannt sein.

1.2 Ziele der Diplomarbeit

Die WCET ist Dreh- und Angelpunkt des C-Compilers WCC (WCET-Aware C-Compiler), welcher am Lehrstuhl 12 für Eingebettete Systeme an der TU Dortmund entwickelt wird [Falk u.a. 2006, Lokuciejewski 2005]. Beim WCC handelt es sich um einen C-Compiler für den Infineon TriCore-Prozessor. Der WCC zeichnet sich durch eine integrierte WCET-Analyse aus. Die WCET wird dabei während des Übersetzungsvorgangs ermittelt. Für die WCET-Analyse wird das Programm aiT der Firma AbsInt Angewandte Informatik GmbH eingesetzt [AbsInt 2008].

Wenn die ermittelte WCET die von außen vorgegebene Zeit überschreitet, bieten sich dem Entwickler folgende Möglichkeiten an: Der Entwickler könnte eine leistungsstärkere Hardware einsetzen. Das geht aber zu Lasten der Kosten eines Produktes. Eine weitere Möglichkeit bietet die manuelle WCET-Minimierung. Die manuelle WCET-Minimierung erfordert jedoch kognitiven Aufwand vom Entwickler und beschreibt einen Trial-and-Error-Vorgang, bei dem der Entwickler die Software manuell umschreibt, erneut kompiliert und daraufhin die WCET neu bestimmt. Dieser zeitaufwändige und fehleranfällige Prozess wird solange durchgeführt, bis die von außen vorgegebene Zeit eingehalten

werden kann. Es wird deutlich, dass ein Automatisieren dieses Prozesses in Form von Compiler-Optimierungen wünschenswert ist.

Leider gibt es bis heute nur wenige WCET-fähige Compiler-Optimierungen. Die Diplomarbeit beschäftigt sich daher mit der Entwicklung von WCET-fähigen Compiler-Optimierungen für den WCC, die detaillierte WCET-Informationen aus einer Assembler-ähnlichen Zwischendarstellung ausnutzen. Eine solche interne Zwischendarstellung wird Low-Level Intermediate Representation (im Folgenden Low-Level IR) genannt und beschreibt eine Repräsentation von Maschinen-Code. Die Transformation einer Low-Level IR in Assembler-Code ist einfach. Auf der anderen Seite gibt es auch interne Zwischendarstellungen, die Hochsprachen-Konstrukte unterstützen. Eine solche Zwischendarstellung wird High-Level Intermediate Representation (im Folgenden High-Level IR) genannt. Weil die High-Level IR sehr nah am Quellcode einer höheren Programmiersprache orientiert ist, ist eine Rücktransformation der High-Level IR in Quellcode einfach.

Detaillierte WCET-Informationen werden durch die WCET-Analyse gewonnen und stehen zu Beginn der Diplomarbeit nur in der Low-Level IR zur Verfügung. Detaillierte WCET-Informationen bezeichnen WCET-Daten, die an Basis-Blöcken angeheftet sind. Basis-Blöcke verwalten mehrere Anweisungen in der Reihenfolge ihrer Abarbeitung und werden im nächsten Kapitel genauer erläutert. Das erste Ziel der Diplomarbeit ist daher die Transformation von detaillierten WCET-Informationen von der Low-Level IR in die High-Level IR. Dieser Vorgang wird im Folgenden **Back-annotation** genannt. Ein weiteres Ziel der Diplomarbeit ist die Ausnutzung dieser WCET-Informationen für die bereits bestehenden High-Level Compiler-Optimierungen **Function Inlining** und **Loop Unrolling**. Die Standard-Optimierungen Function Inlining und Loop Unrolling werden üblicherweise zur Reduktion der durchschnittlichen Programm-Laufzeit (Average-Case Execution Time, ACET) eingesetzt und sollen in dieser Arbeit mit der neuen Zielfunktion der *Minimierung der maximalen Programm-Laufzeit* betrachtet werden. Im letzten Schritt sollen die neuen WCET-gesteuerten Optimierungen an Realworld-Benchmarks des WCCs evaluiert werden.

Für die Optimierung Function Inlining wird ein Verfahren aus dem Bereich des *maschinellen Lernens* eingesetzt, um erstmals für den WCC automatisch eine Heuristik für eine Optimierung zu *generieren*. Die generierte Heuristik soll im Rahmen dieser Arbeit die Frage beantworten, wann Function Inlining durchgeführt werden soll und wann nicht. Für diesen Zweck wird das relativ neue Verfahren Zufallswälder (Random Forests) des Statistikers Leo Breiman eingesetzt [Breiman 2001]. Zufallswälder klassifizieren auf der Basis vieler Entscheidungsbäume und versuchen dadurch die Vorhersage zu verbessern.

Beim Loop Unrolling ist neben der Ausnutzung von WCET-Informationen der Einsatz der Schleifenanalyse des WCCs zentral. Die Schleifenanalyse, die von Daniel Cordes entwickelt wurde [Cordes 2008] und auf Basis der Abstrakten Interpretation arbeitet, wird erstmals in dieser Arbeit für eine Optimierung produktiv eingesetzt. Sie soll mehr Potential für die neue WCET-gesteuerte Optimierung Loop Unrolling schaffen, indem sie auch komplizierte Schleifen für die Optimierung zugänglich macht, die mit dem bisherigen Verfahren nicht analysiert werden können.

1.3 Verwandte Arbeiten

Die Transformation von WCET-Informationen von der Low-Level IR in die High-Level IR ist vergleichbar mit der Arbeit von Daniel Schulte. Daniel Schulte befasste sich in seiner Diplomarbeit mit dem Thema „Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler“ [Schulte 2007]. Zusätzlich benötigte Kontrollflussinformationen zur WCET-Berechnung, wie z.B. Schleifeniterationsgrenzen, werden in seiner Arbeit auf hoher Abstraktionsebene modelliert und dann in niedrigere Abstraktionsebenen transformiert. Dabei werden auch Techniken zur Konsistenzsicherung dieser Kontrollflussinformationen für die Übersetzungen und Optimierungen des Programms im Compiler zur Verfügung gestellt. Auch Daniel Schulte musste zwischen der High-Level IR und Low-Level IR einen Informationsaustausch gewährleisten. Im Vergleich zur Back-annotation jedoch erfolgt der Informationsfluss in umgekehrter Richtung (von hoher Abstraktionsebene in niedrigere Abstraktionsebenen).

Es existieren viele verwandte Arbeiten, die sich mit dem Thema pfadgesteuerte Optimierungen beschäftigen. Dabei wird meistens der häufig ausgeführte Pfad (Average-Case Execution Path, ACEP) optimiert. So werden in [Chang u.a. 1991] Profil-Informationen durch mehrmaliges Ausführen eines Programms für verschiedene Eingaben gewonnen. Anschließend benutzen viele Standard-Optimierungen wie z.B. Constant Propagation oder Common Subexpression Elimination die Profil-Informationen zur Reduktion der durchschnittlichen Programm-Laufzeit. Profilbasierte Optimierungen besitzen allerdings den Nachteil, dass auf neuen Eingabedaten der Wirkungsgrad der Optimierung abnimmt.

Auch in [Cohn u. Lowney 2000] werden Profil-Informationen zur Reduktion der ACET eingesetzt. Für diesen Zweck werden u.a. Standard-Optimierungen wie z.B. Function Inlining auf dem häufig ausgeführten Pfad ausgeführt. Außerdem wird beschrieben, wie Profil-Informationen bei der Register-Allokation ausgenutzt werden können. Die Register-Allokation stellt eine wichtige Phase für den Compiler dar und wird in Kapitel 3 vorgestellt.

Einen neuen Forschungsbereich stellen Optimierungen dar, die den längsten Ausführungspfad eines Programms (Worst-Case Execution Path, WCEP) verbessern. Daher gibt es nur wenige Arbeiten, die sich mit diesem Thema beschäftigen.

In [Zhao u.a. 2006] werden WCET-Informationen durch einen in den Compiler integrierten Timing-Analyzer zur Verfügung gestellt. Der Timing-Analyzer kann die WCET aller Pfade einer Funktion bestimmen. Eine WCET-Minimierung wird durch Transformationen der Low-Level IR mit Hilfe von Low-Level Compiler-Optimierungen wie das Duplizieren von Programmteilen oder Loop Unrolling erreicht.

Auch [Puaut 2006] und [Lokuciejewski u.a. 2008a] arbeiten auf der Low-Level IR-Ebene und nutzen Speicherhierarchien aus, um eine WCET-Minimierung zu erreichen. In [Puaut 2006] wird ein Algorithmus vorgestellt, der statisch ausgewählte Teile eines Programms in den Cache lädt und anschließend durch einen *Lockdown* vor Verdrängung schützt. Dies ermöglicht eine bessere Abschätzung der WCET. In [Lokuciejewski u.a. 2008a] wird die WCET-gesteuerte Optimierung Procedure Positioning

vorgestellt. Die Optimierung führt auf Basis eines WCET-basierten Graphen für Funktionsaufrufe eine Umordnung von Funktionen im Speicher durch, sodass es zu weniger Cache-Fehlzugriffen kommt. Dabei werden Funktionen, die sich häufig gegenseitig aufrufen, nebeneinander positioniert.

Im Gegensatz zu den obigen Arbeiten wird in dieser Diplomarbeit eine WCET-Minimierung durch High-Level Compiler-Optimierungen angestrebt. Die dem Autor einzig bekannte Arbeit, die sich mit diesem Thema beschäftigt [Lokuciejewski u.a. 2008b], ist am Lehrstuhl 12 für Eingebettete Systeme an der TU Dortmund entstanden. Darin wird beschrieben, wie die High-Level Compiler-Optimierung Procedure Cloning eine präzise Annotation von Schleifeniterationsgrenzen ermöglicht. Präzise Schleifeniterationsgrenzen führen zu einer besseren WCET-Abschätzung. Andererseits wird durch die Optimierung eine Code-Vergrößerung verursacht, die der Autor durch einen Parameter begrenzt.

1.4 Aufbau der Diplomarbeit

Die Diplomarbeit ist wie folgt aufgebaut:

- **Kapitel 2** bietet eine kurze Einführung in die WCET-Analyse. Unterschiedliche Analyseverfahren werden betrachtet. Außerdem wird das Programm aiT der Firma AbsInt Angewandte Informatik GmbH näher vorgestellt.
- **Kapitel 3** beschäftigt sich mit dem WCET-optimierenden C-Compiler WCC. Der Aufbau des WCCs wird präsentiert und die Situation vor der Diplomarbeit wird erläutert.
- **Kapitel 4** behandelt das Thema Back-annotation: Transformation von WCET-Informationen von der Low-Level IR in die High-Level IR.
- **Kapitel 5** beschreibt ein neues Verfahren mit dem Namen Cold Path. Dabei handelt es sich um eine zusätzliche Auswertung der WCET-Informationen, die nach der Back-annotation-Phase auf High-Level IR-Ebene vorhanden sind.
- **Kapitel 6** befasst sich mit dem maschinellen Lernen einer Heuristik für die High-Level Compiler-Optimierung Function Inlining. Beim Lernen werden auch WCET-Informationen, die nach der Back-annotation-Phase zur Verfügung stehen, ausgenutzt.
- **Kapitel 7** hingegen zeigt die Entwicklung einer WCET-gesteuerten Loop Unrolling-Optimierung. Anders als in Kapitel 6 wird die Heuristik manuell erzeugt. In der Optimierung werden sowohl WCET- und Cold Path-Informationen als auch die Ergebnisse der statischen Schleifenanalyse des WCCs ausgenutzt.
- **Kapitel 8** demonstriert die Entstehung und Evaluation von Ergebnissen.
- **Kapitel 9** enthält eine Zusammenfassung der Diplomarbeit und gibt einen Ausblick auf mögliche weitere Arbeiten.

2 WCET-Analyse

Der Begriff Worst-Case Execution Time (WCET) bezeichnet die maximale Laufzeit eines Programms. Damit wird die maximale Laufzeit eines Programms über alle denkbaren Eingabedaten hinweg gemeint. Die Ermittlung der WCET ist im Allgemeinen nicht berechenbar, weil sich das Problem der Berechnung der WCET eines Programms auf das Halteproblem wie folgt reduzieren lässt: Zuerst wird die WCET eines zu analysierenden Programms ermittelt. Anschließend lässt sich durch einen Vergleich in $O(1)$ entscheiden, ob die berechnete WCET endlich ist oder nicht ($WCET < \infty$). Aus diesem Grund berechnen WCET-Analyseverfahren, die im weiteren Verlauf dieses Kapitels vorgestellt werden, nur eine approximative Lösung für die echte WCET. Im Folgenden bezeichnet WCET_{real} die echte WCET, die im Allgemeinen unbekannt ist. WCET_{est} hingegen bezeichnet die abgeschätzte WCET. Aus diesem Grund werden WCET-Informationen, die im WCET-fähigen Compiler WCC nach der WCET-Analyse verfügbar sind, in den nachfolgenden Kapiteln WCET_{est}-Informationen genannt. Der WCC wird im nächsten Kapitel ausführlich beschrieben.

Neben der WCET gibt es auch die ACET (Average-Case Execution Time) und BCET (Best-Case Execution Time). In Kapitel 1 wurde erwähnt, dass die ACET für die durchschnittliche Programm-Laufzeit steht. Bei der ACET handelt es sich um eine typische Zielfunktion, die in vielen optimierenden Compilern, wie z.B. im C-Compiler gcc [GCC 2008], benutzt wird. Die BCET hingegen ist die kürzeste Ausführungsdauer eines Programms. Somit werden die möglichen Ausführungszeiten des zu analysierenden Programms durch die Werte von BCET_{real} und WCET_{real} eingegrenzt (vgl. Abb. 2.1).

Abbildung 2.1: *Ausführungszeiten von Programmen*

Abschnitt 2.1 stellt ingesamt drei verschiedene Verfahren zur Berechnung der WCET_{est} vor. Zuvor werden jedoch die Anforderungen an solche Verfahren diskutiert. Anschließend wird in Abschnitt 2.2 das kommerzielle Programm aiT, welches eines der vorgestellten Verfahren umsetzt, näher erläutert. Dieses Programm ist für die vorliegende Arbeit von Bedeutung, weil es im WCC für eine WCET-Analyse eingesetzt wird.

2.1 Verfahren zur Berechnung der WCET$_{est}$

Sicherheit und Präzision sind zwei wichtige Anforderungen, die an Verfahren zur Berechnung der WCET$_{est}$ gestellt werden. Eine Approximation der echten WCET heißt **sicher**, wenn garantiert werden kann, dass die geschätzte WCET mindestens so groß ist, wie die echte WCET:

$$WCET_{est} \geq WCET_{real} \tag{2.1}$$

Kann die Gültigkeit dieser Ungleichung nicht garantiert werden, so kann das betrachtete Programm im schlimmsten Fall nicht alle vorgegebenen Zeitschranken einhalten. Dies kann bei einem harten Realzeit-System zu einer Katastrophe führen.

Außerdem muss sichergestellt werden, dass die Approximation der echten WCET möglichst **präzise** ist:

$$WCET_{est} - WCET_{real} \longrightarrow min \tag{2.2}$$

Ansonsten ist die Überabschätzung der echten WCET zu groß und der Einsatz der gewonnenen Ergebnisse in der Praxis aus wirtschaftlichen Gründen nicht sinnvoll.

Ein Verfahren zur Berechnung der WCET$_{est}$, das diesen Anforderungen nicht gerecht wird, ist in der Praxis nur bedingt einsetzbar. Daher sollten solche Verfahren eine sichere und möglichst präzise Abschätzung der WCET$_{real}$ zum Ziel haben. Im Folgenden werden statische, dynamische und hybride Verfahren zur Berechnung der WCET$_{est}$ vorgestellt. Diese betrachten den Assembler- bzw. Maschinen-Code des zu analysierenden Programms, um eine möglichst präzise Abschätzung der WCET$_{real}$ angeben zu können.

2.1.1 Statische WCET-Analyse

Bei der statischen WCET-Analyse wird das zu analysierende Programm zu keinem Zeitpunkt der Analyse auf einer bestimmten Zielarchitektur ausgeführt. Stattdessen wird der Maschinen-Code des Programms mit Hilfe eines statischen Verfahrens analysiert, um so eine Abschätzung der WCET$_{real}$ zu ermitteln. Für diesen Zweck wird zuerst der Kontrollflussgraph (Control Flow Graph, CFG) des zu analysierenden Programms aufgebaut. Der Kontrollflussgraph $CFG = (V, E)$ ist ein gerichteter Graph, wobei V die Menge aller Basis-Blöcke und E die Menge aller Kontrollflusskanten ist. Ein Basis-Block ist im Allgemeinen eine maximale Sequenz von Anweisungen, die nur durch die erste Anweisung betreten wird und durch die letzte Anweisung verlassen wird.

Der Kontrollflussgraph ist die zentrale Datenstruktur der statischen WCET-Analyse und dient zur Beschreibung der möglichen Ausführungspfade des Programms.

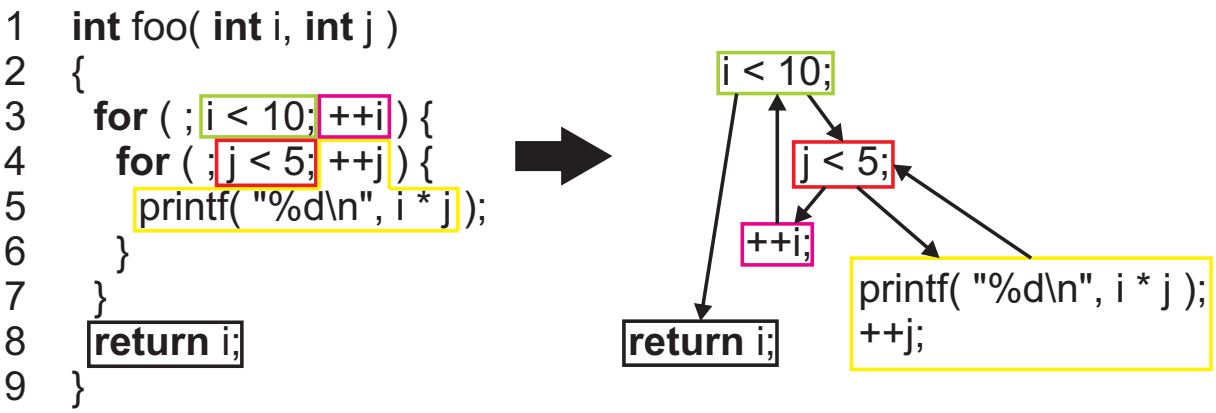


Abbildung 2.2: *Beispiel für einen Kontrollflussgraphen*

In Abb. 2.2 ist der zugehörige Kontrollflussgraph einer Funktion abgebildet. Es wird ersichtlich, dass die gerichteten Kanten zwischen Basis-Blöcken die möglichen Ausführungspfade beschreiben.

Nachdem der Kontrollflussgraph des Programms aufgebaut wurde, wird die $WCET_{est}$ der einzelnen Basis-Blöcke bestimmt. Aufgrund der Unentscheidbarkeit der WCET-Berechnung werden neben dem zu analysierenden Programm weitere Informationen benötigt. Dabei handelt es sich um zusätzliche Informationen über den Kontrollfluss des zu analysierenden Programms. Diese geben z.B. Auskunft über maximale Iterationshäufigkeit von Schleifen und sind für eine WCET-Analyse zwingend notwendig.

Außerdem muss die statische WCET-Analyse über ein sehr genaues Modell der betrachteten Zielarchitektur verfügen. So erfordert z.B. die Analyse von Speicherzugriffen ein sehr genaues Cache-Modell, um die Kosten eines jeden Speicherzugriffs möglichst genau abschätzen zu können. Zudem beinhalten moderne Prozessoren eine ausgeklügelte Pipeline zur parallelen Abarbeitung von Maschinen-Code. Um eine möglichst präzise Abschätzung der $WCET_{real}$ durchführen zu können, muss auch die Prozessor-Pipeline modelliert und bei der Berechnung berücksichtigt werden.

Im letzten Schritt der WCET-Analyse wird auf Basis des um $WCET_{est}$-Informationen erweiterten Kontrollflussgraphen der WCEP berechnet. Dabei kommt ein bestimmtes Berechnungsverfahren wie z.B. Implicit Path Enumeration Technique (IPET) zum Einsatz. Dieses Verfahren berechnet mit Hilfe von Integer Linear Programming (ILP) die $WCET_{est}$ des zu analysierenden Programms, indem es ein Optimierungsproblem aufstellt und anschließend löst. Neben dem Berechnungsverfahren IPET gibt es noch andere Berechnungsverfahren wie z.B. die Baum- oder Pfad-basierte Berechnung, auf die aber nicht weiter eingegangen werden soll.

Das Implementieren eines genauen Modells für eine bestimmte Zielarchitektur ist sehr schwierig und kostet viel Zeit. Dafür kann aber die statische WCET-Analyse eine sichere und möglichst präzise Approximation der $WCET_{real}$ gewährleisten.

2.1.2 Dynamische WCET-Analyse

Anders als bei der statischen WCET-Analyse wird bei der dynamischen WCET-Analyse das zu analysierende Binär-Programm auf der betrachteten Zielarchitektur mehrmals mit verschiedenen Eingaben ausgeführt. Die dynamische WCET-Analyse versucht anschließend aus den Ergebnissen der einzelnen Testläufe auf die WCET_{real} des Programms zu schließen. Meistens ist jedoch die Eingabe, die zu der längsten Programm-Laufzeit führt, nicht bekannt. Zudem können bei komplexen Programmen nicht alle möglichen Eingaben in endlicher Zeit getestet werden. Es wird schnell klar, dass mit Hilfe eines dynamischen Verfahrens keine sichere Approximation der WCET_{real} garantiert werden kann. Im Gegensatz zu der statischen WCET-Analyse jedoch erfordert die dynamische WCET-Analyse kein präzises Modell der Zielarchitektur. Der große Vorteil der dynamischen WCET-Analyse ist unzweifelhaft die einfache und schnelle Umsetzung des Verfahrens in der Praxis. Allerdings müssen die Eingabedaten für die einzelnen Testläufe gut überlegt sein, damit ein möglichst repräsentatives Ergebnis erzielt werden kann.

2.1.3 Hybride WCET-Analyse

Die hybride WCET-Analyse versucht die Vorteile der beiden vorgestellten Verfahren zu kombinieren. Wie die dynamische WCET-Analyse verfügt auch die hybride WCET-Analye über kein präzises Modell der betrachteten Zielarchitektur.

Bei der hybriden WCET-Analyse wird ein Kontrollflussgraph des zu analysierenden Programms aufgebaut und durch Simulationen werden Ergebnisse für einzelne Teilpfade im Kontrollflussgraph gewonnen. Dies macht den dynamischen Anteil der hybriden WCET-Analyse aus.

Anschließend wird z.B. über ILP der längste Pfad im Kontrollflussgraph bestimmt. Daraus lässt sich der statische Charakter des Verfahrens erkennen.

2.2 aiT – Programm zur Berechnung der WCET_{est}

In Kapitel 1 wurde betont, dass im WCC das Programm aiT der Firma AbsInt Angewandte Informatik GmbH zum Einsatz kommt [AbsInt 2008]. Dabei handelt es sich um ein Programm, welches das vorgestellte Verfahren der statischen WCET-Analyse benutzt, um die WCET_{real} eines anderen Programms abzuschätzen. Im Folgenden wird die Funktionsweise des Programms aiT anhand der Abb. 2.3 Schritt für Schritt erklärt.

Das Programm aiT erhält als Eingabe ein zu analysierendes Binär-Programm. In Abschnitt 2.1.1 wurde beschrieben, dass eine statische WCET-Analyse neben dem zu analysierenden Binär-Programm zusätzliche Informationen benötigt, die für eine statische WCET-Analyse zwingend notwendig sind. Aus diesem Grund erhält aiT als weitere Eingabe eine Datei mit Benutzer-Annotationen. Diese geben z.B. Auskunft über maximale Iterationshäufigkeit von Schleifen und maximale Rekursionstiefen.

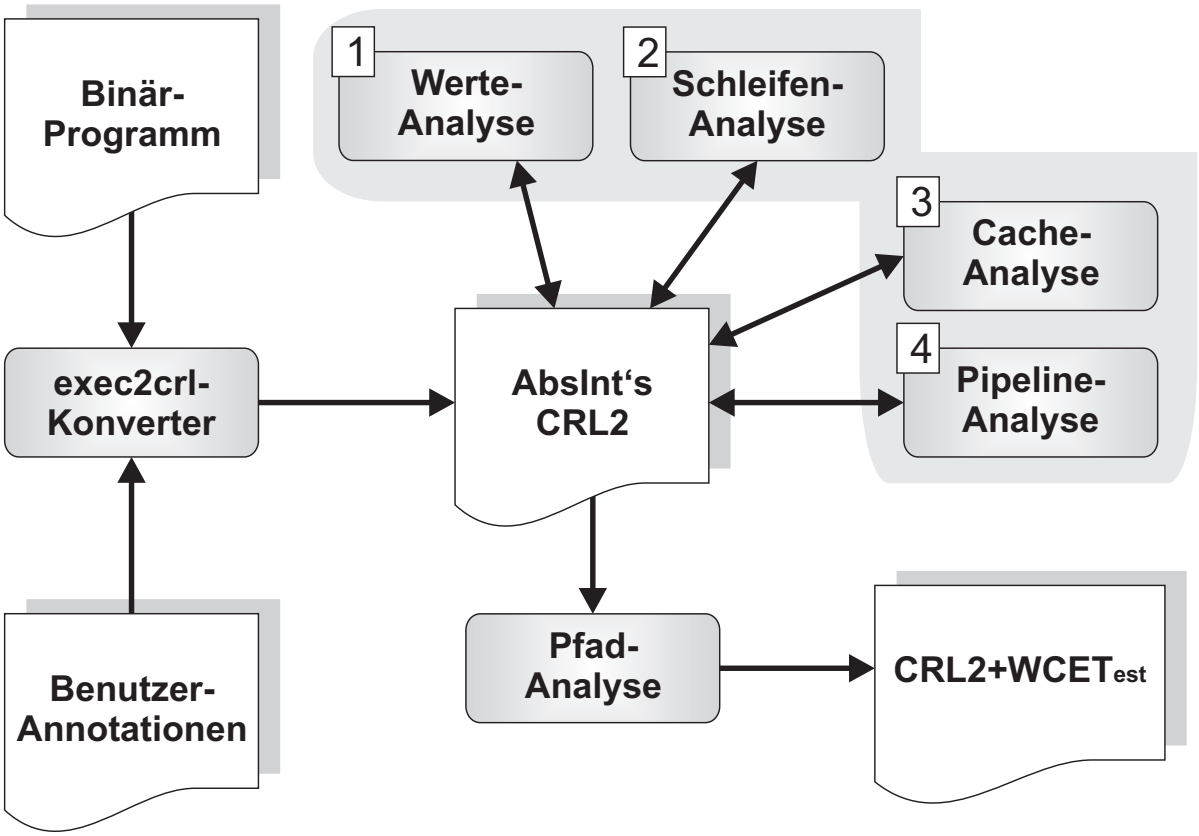


Abbildung 2.3: *Funktionsweise des Programms aiT*

Das zu analysierende Binär-Programm wird zuerst mittels des exec2crl-Konverters disassembliert und in die aiT-interne Datenstruktur CRL2 (Control Flow Representation Language) überführt. Anschließend beginnt die statische WCET-Analyse. Die statische WCET-Analyse besteht aus mehreren Phasen, die im Folgenden kurz erläutert werden:

- Die *Werte-Analyse* ermittelt für jeden Ausführungszeitpunkt des zu analysierenden Binär-Programms die möglichen Werte von Prozessor-Registern, um z.B. die Adressen von Speicherzugriffen zu bestimmen.
- Die *Schleifen-Analyse* ermittelt für sehr einfache Schleifen die minimale und maximale Iterationshäufigkeit von Schleifen.
- In der *Cache-Analyse* wird mit Hilfe eines formalen Cache-Modells jeder Speicherzugriff in *Cache-Hit*, *Cache-Miss* oder *unknown* klassifiziert.
- In der *Pipeline-Analyse* wird ein genaues Modell der Pipeline benutzt, um das Verhalten des Programms in der Pipeline des Zielprozessors zu simulieren. Dabei werden auch die Resultate der Cache-Analyse benutzt. Nach der Pipeline-Analyse ist die $WCET_{est}$ eines jeden CRL2-Basis-Blocks bekannt.

Es wird deutlich, dass das Programm aiT über ein sehr genaues Modell der betrachteten Zielarchitektur verfügt. Dadurch wird eine möglichst präzise Approximation der $WCET_{real}$ gewährleistet.

Im Gegensatz zu den vorgestellten Analysen basiert die *Pfad-Analyse* von aiT nicht auf den Theorien der Abstrakten Interpretation. Die Pfad-Analyse modelliert mit Hilfe von ILP alle möglichen Ausführungspfade des zu analysierenden Binär-Programms und bestimmt letztendlich den längsten Ausführungspfad (WCEP) im Programm, dessen Länge der WCET_{est} entspricht. Die Pfad-Analyse benutzt dabei das Konzept der Kontexte, um die Genauigkeit bei der Abschätzung der WCET_{real} zu erhöhen. Die WCET-Analyse von aiT unterstützt Kontexte für Funktionen und Schleifen. Funktionskontexte erlauben eine getrennte Betrachtung der Funktionsaufrufe einer bestimmten Funktion, wohingegen Schleifenkontexte eine getrennte Betrachtung der Schleifeniterationen einer Schleife ermöglichen. Somit kann aiT Bedingungen in Abhängigkeit eines bestimmten Kontextes auswerten und die möglichen Werte von Prozessor-Registern weiter einschränken. Dies führt in der Regel zu einer Verbesserung der Approximation. Im Folgenden wird das um Kontexte erweiterte Berechnungsmodell von aiT kurz vorgestellt.

Weil ein Basis-Block mehrere Nachfolger haben kann, werden auch die Kontrollflusskanten im Kontrollflussgraph mit WCET_{est}-Informationen versehen. Sei $T(e, c)$ die WCET_{est} einer Kontrollflusskante e, die in einem bestimmten Kontext c gültig ist. Des Weiteren sei $C(e, c)$ die Ausführungshäufigkeit der Kontrollflusskante e im Kontext c. Dann ergibt sich zur Berechnung der WCET_{est} folgendes Optimierungsproblem:

$$\sum_{\forall e,c} C(e, c) * T(e, c) \rightarrow max \tag{2.3}$$

Die Nebenbedingungen des Optimierungsproblems ergeben sich insbesondere aus den Benutzer-Annotationen, die dem Programm aiT in Form einer Datei zur Verfügung gestellt werden. Somit kann aiT durch das Lösen eines Optimierungsproblems eine Approximation der WCET_{real} erreichen.

3 WCC – WCET-optimierender C-Compiler

Anfangs wurde die Programmierung von Eingebetteten Systemen direkt in Assembler durchgeführt. Heute jedoch wird immer öfter aufgrund der ständig wachsenden Komplexität von Eingebetteten Systemen eine höhere Programmiersprache wie C bevorzugt. Die Aufgabe der Übersetzung einer höheren Programmiersprache in (optimierten) Assembler-Code übernehmen Compiler.

Zu den wichtigsten Anforderungen an Compiler für Eingebettete Systeme gehört sicherlich die möglichst hohe Code-Qualität. Eine möglichst hohe Code-Qualität kann sich z.B. in Form von Laufzeit-Effizienz, geringem Energieverbrauch, geringer Code-Größe oder Realzeit-Fähigkeit ausdrücken. In Abschnitt 1.1 wurde betont, dass das Einhalten von Zeitschranken eine wichtige Anforderung an Eingebettete Systeme, die zugleich Realzeit-Systeme sind, ist, weswegen die Realzeit-Fähigkeit eine wichtige Anforderung an Compiler für Eingebettete Systeme darstellt, die aber oftmals unberücksichtigt bleibt.

Diesen Anforderungen versucht der C-Compiler WCC (WCET-Aware C-Compiler), der am Lehrstuhl 12 für Eingebettete Systeme an der TU Dortmund entwickelt wird [Falk u.a. 2006], gerecht zu werden. Der WCC wird im nächsten Abschnitt näher vorgestellt.

3.1 WCC

Zielarchitektur für den WCC ist der Infineon TriCore-Prozessor. Dabei handelt es sich um einen 32-bit digitalen Signalprozessor (DSP). Digitale Signalprozessoren haben besondere Eigenschaften wie separate Adress- und Datenregister und Multiply-Accumulate-Befehle (MAC), und eignen sich besonders für den Einsatz in Eingebetteten Systemen.

Der WCC unterstützt den C99-Standard [ISO/IEC 9899:1999 (E)] und zeichnet sich durch eine integrierte WCET-Analyse aus. Für diesen Zweck wurde das kommerzielle Programm aiT der Firma AbsInt Angewandte Informatik GmbH in den WCC integriert [AbsInt 2008]. Dies ermöglicht eine automatische Ermittlung der WCET_{est} während des Übersetzungsvorgangs. Die so gewonnene WCET_{est} kann anschließend durch WCET-Optimierungen ausgenutzt werden.

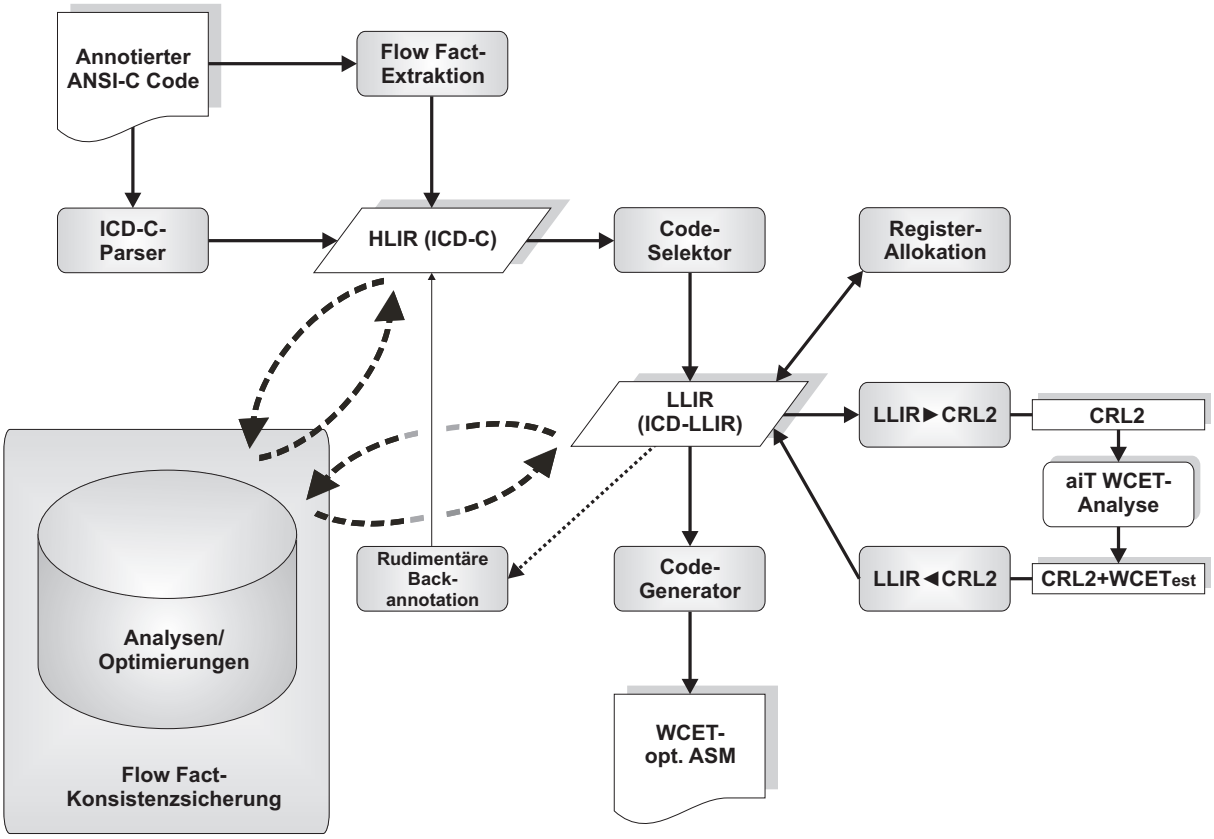


Abbildung 3.1: *WCC: Compileraufbau vor Beginn der Diplomarbeit*

In Abb. 3.1 ist der Compileraufbau des WCCs vor Beginn der Diplomarbeit zu sehen.

Der WCC erhält als Eingabe den annotierten ANSI-C Code eines Programms, das aus mindestens einer C-Datei besteht. Der ANSI-C Code enthält in Form von Pragma-Direktiven zusätzliche Kontrollflussinformationen, die zur WCET-Berechnung notwendig sind. Zusätzlich benötigte Kontrollflussinformationen sind Informationen wie die maximale Anzahl von Schleifeniterationen und werden aus dem annotierten ANSI-C Code durch den **Flow Fact-Manager** extrahiert und in **ICD-C**, eine High-Level IR, überführt. Der ANSI-C Code wird durch einen C99-Parser ebenfalls in ICD-C übersetzt. Die ICD-C wird in Abschnitt 3.1.1 detailliert vorgestellt. Abschnitt 3.1.3 hingegen geht näher auf den Flow Fact-Manager ein.

Die Übersetzung der ICD-C in die Low-Level IR **ICD-LLIR** wird durch den **Code-Selektor** durchgeführt. Die ICD-LLIR wird in Abschnitt 3.1.2 und der Code-Selektor in Abschnitt 3.1.4 näher erläutert.

Damit am Ende ausführbarer Maschinen-Code erzeugt werden kann, muss an dieser Stelle noch die **Register-Allokation** aufgerufen werden. In Abschnitt 3.1.5 wird erklärt, was es mit der Register-Allokation auf sich hat.

Anschließend kann die ICD-LLIR automatisch um WCET_{est}-Informationen erweitert werden. Für diesen Zweck muss die ICD-LLIR-Datenstruktur zuerst in die aiT-interne Datenstruktur CRL2 kon-

vertiert werden. Das ist die Aufgabe des LLIR→CRL2-Konverters. Erst dann kann das Programm aiT eine WCET-Analyse durchführen. Damit die $WCET_{est}$-Informationen in die ICD-LLIR-Datenstruktur importiert werden können, muss in einem letzten Schritt der CRL2→LLIR-Konverter aufgerufen werden. Eine detaillierte Beschreibung der Einbindung der WCET-Analyse ist in [Lokuciejewski 2005] zu finden.

Nach dem Importieren der $WCET_{est}$-Informationen in die ICD-LLIR-Datenstruktur kann die Back-annotation durchgeführt werden. Die Back-annotation transformiert die $WCET_{est}$-Informationen, die nun in der ICD-LLIR enthalten sind, in die ICD-C. In Abb. 3.1 ist der Pfeil zwischen dem Back-annotation Modul und der ICD-C dünn dargestellt, weil zu Beginn der Diplomarbeit noch keine detaillierten $WCET_{est}$-Informationen transformiert werden. Abschnitt 3.1.6 geht näher auf die Back-annotation-Phase ein.

Der Code-Generator erzeugt anschließend aus der ICD-LLIR ausführbaren Assembler-Code.

Wie jeder moderne Compiler auch setzt der WCC viele Analyse- und Optimierungsverfahren ein, um den Anforderungen an Compiler für Eingebettete Systeme gerecht zu werden. Eine Voraussetzung zur Code-Optimierung ist das Vorhandensein einer internen Zwischendarstellung von Code, die Code-Manipulationen ermöglicht und notwendige Analysen für Optimierungen bereitstellt [Falk WS–2007/2008].

Die Zwischendarstellungen ICD-C und ICD-LLIR werden im Folgenden näher betrachtet, weil sie im WCC zum Einsatz kommen.

3.1.1 ICD-C

Die High-Level IR ICD-C (kurz IR) ist eine Zwischendarstellung, die auf hoher Abstraktionsebene angesiedelt ist. Die ICD-C wurde vom Informatik Centrum Dortmund [ICD 2008] entwickelt und enthält komplexe Kontroll- und Datenflusskonstrukte wie Schleifen (`for`, `while`, `do-while`), Verzweigungen (`if`, `if-else`, `switch`) oder Array-Zugriffe (Index-Operator `[]`) der Programmiersprache C. Weil die ICD-C sehr nah am Quellcode orientiert ist, ist die Rücktransformation dieser Zwischendarstellung in Quellcode einfach.

3.1.1.1 ICD-C-Aufbau

Abb. 3.2 auf der nächsten Seite zeigt den Aufbau der ICD-C.

Das gesamte zu analysierende C-Programm wird in ICD-C durch ein Objekt der Klasse IR repräsentiert. Die IR verwaltet als zentrale Instanz alle Übersetzungseinheiten (IR_CompilationUnit) eines C-Programms und deren globale Symbole. Ein C-Programm besteht normalerweise aus mehreren Übersetzungseinheiten. Eine Übersetzungseinheit entspricht einer C-Datei. Die Übersetzungseinheiten selbst bestehen aus Funktionen (IR_Function). Der Funktionsrumpf einer Funktion wird durch einen

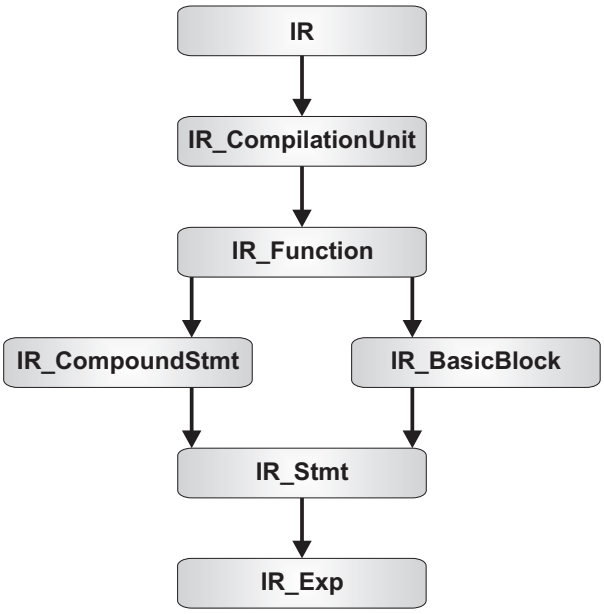


Abbildung 3.2: *ICD-C-Aufbau*

Anweisungsblock (IR_CompoundStmt) repräsentiert. Eine solche Anweisung beinhaltet alle Anweisungen (IR_Stmt) der Funktion und wird auch *Top Compound Statement* genannt. Eine Funktion kann aber auch als eine Liste von IR-Basis-Blöcken (IR_BasicBlock) angesehen werden. Die IR-Basis-Blöcke ermöglichen wiederum den Zugriff auf die einzelnen Anweisungen der Funktion. Um dem Sprachumfang von C Rechnung zu tragen, gibt es in ICD-C viele Konkretisierungen der abstrakten Klasse IR_Stmt:

- IR_AsmStmt entspricht einer Assembler-Anweisung, die im C-Quellcode enthalten ist.
- IR_CompoundStmt beschreibt einen Anweisungsblock, in der Anweisungen in der Reihenfolge ihrer Abarbeitung verwaltet werden.
- IR_ExpStmt repräsentiert eine Ausdrucks-Anweisung, die sich üblicherweise aus mehreren Ausdrücken zusammensetzt und somit einen Ausdrucks-Baum darstellt.
- IR_JumpStmt verkörpert unbedingte Sprünge im Quellcode (`break`, `continue`, `goto`, `return`).
- IR_LoopStmt stellt Schleifen im Quellcode dar (`for`, `while`, `do-while`).
- IR_SelectionStmt ist eine bedingte Verzweigung (`if`, `if-else`, `switch`).
- ...

Abb. 3.3 zeigt eine Übersicht über alle Klassen, die von der Klasse IR_Stmt abgeleitet sind. Dabei stellen die unterstrichenen Klassen abstrakte Klassen dar.

Eine Anweisung setzt sich üblicherweise aus mehreren Ausdrücken zusammen. Ausdrücke werden in ICD-C durch die abstrakte Klasse IR_Exp repräsentiert. Um dem C99-Standard gerecht zu werden, bietet die ICD-C diverse Konkretisierungen dieser Klasse:

- IR_BinaryExp beschreibt einen binären Ausdruck (+, -, *, /, ...).

- IR_UnaryExp hingegen beschreibt einen unären Ausdruck (++, --, ...).
- IR_AssignExp repräsentiert eine Zuweisung (=, +=, -=, ...).
- IR_IndexExp entspricht einem Array-Zugriff mittels des Index-Operators (`a[i]`).
- IR_ComponentAccessExp ist zuständig für Komponentenzugriffe (`s.x`, ...).
- IR_CallExp stellt einen Funktionsaufruf dar.
- ...

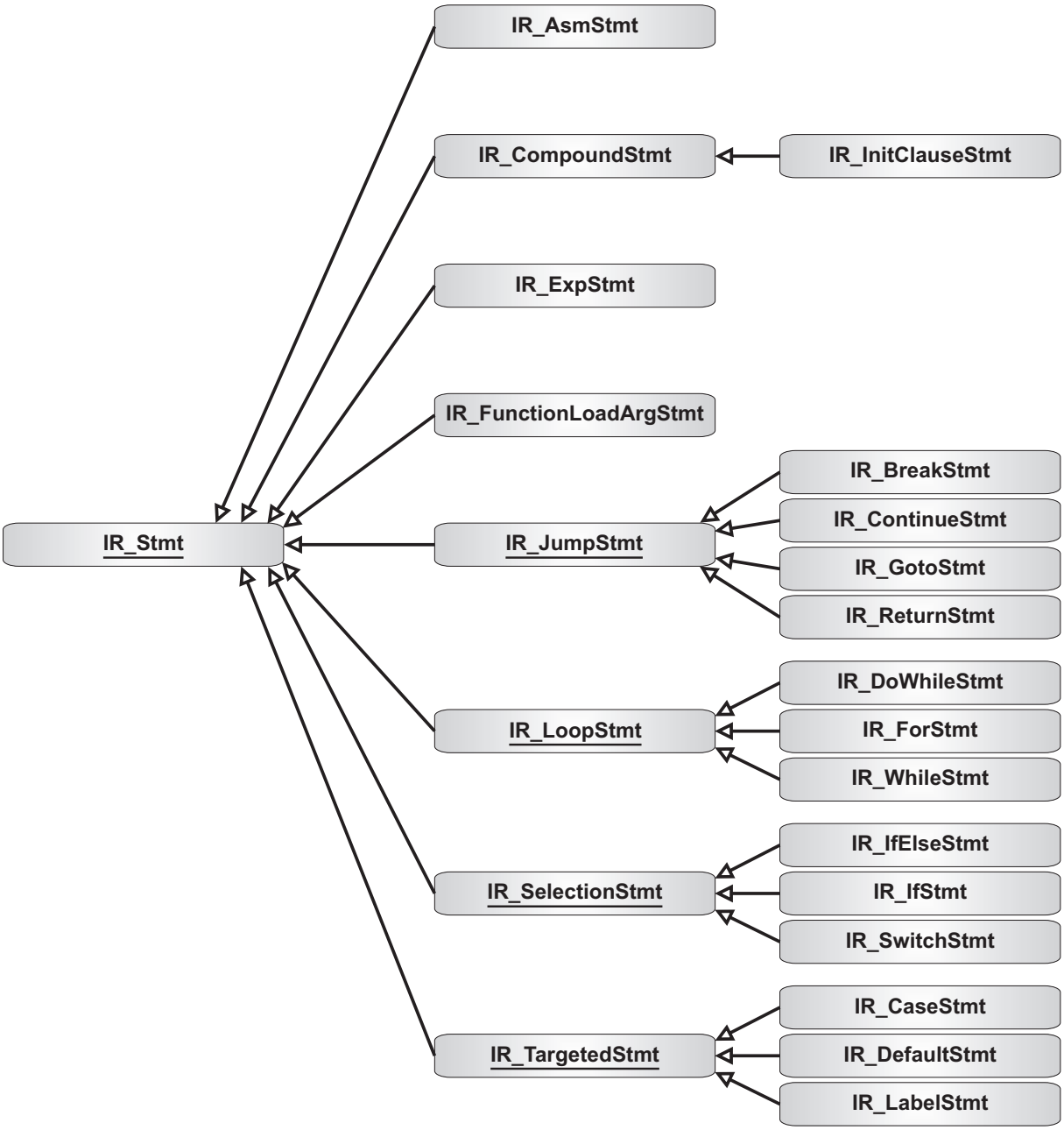


Abbildung 3.3: *ICD-C-Anweisungen*

Zusammengefasst lässt sich sagen, dass die ICD-C den gesamten Sprachumfang der Programmiersprache C nach dem C99-Standard unterstützt. Auch der C89-Standard wird unterstützt. Die ICD-C

ist in [ICD-C 2008] detailliert beschrieben.

3.1.1.2 ICD-C-Analysen

Verschiedene Kontrollfluss- und Datenflussanalysen sind Bestandteil der ICD-C und ermöglichen u.a. ein bequemes Durchlaufen der ICD-C-Datenstruktur. Dies wird mit Hilfe des **Callback-Mechanismus** gewährleistet. Beim Callback-Mechanismus stellt der Benutzer eine (Analyse-)Funktion zur Verfügung, die automatisch aufgerufen wird, wenn ein bestimmtes Ereignis eintritt [Schmaranz 2001]. ICD-C bietet Funktionen, die z.B. über alle Anweisungen eines Anweisungsblocks iterieren und bei jeder Anweisung ein solches Ereignis werfen. Dies ermöglicht ein einfaches Ergänzen von neuen Analysefunktionen.

ICD-C enthält auch eine statische Schleifenanalyse. Der ICD-C Loop Analyzer (IR_LoopAnalyzer) kann für einfache Schleifen die genaue Iterationsanzahl ermitteln. Die Ergebnisse können anschließend durch Optimierungen ausgenutzt werden. Die Optimierung Loop Unrolling ist eine Optimierung, die den ICD-C Loop Analyzer einsetzt und in Kapitel 7 näher betrachtet wird. Weil der ICD-C Loop Analyer nicht alle Schleifen analysieren kann, wird in Kapitel 7 gezeigt, wie zusätzliche Informationsquellen eingesetzt werden können, um mehr Potential für die neuartige WCET-fähige Optimierung Loop Unrolling zu schaffen.

3.1.1.3 ICD-C-Erweiterbarkeit

Benutzerdaten (UserData) vom Typ IR_PersistentObject bieten eine flexible Möglichkeit zur Erweiterung der ICD-C-Datenstruktur. Benutzerdaten können nämlich an jede Klasse der ICD-C-Datenstruktur angeheftet werden, weil jede Klasse von IR_PersistentObject abgeleitet ist. Aus diesem Grund besitzen alle Klassen der ICD-C die folgenden Funktionen:

- void setUserData(ITEM_t key, IR_PersistentObject *data)
- IR_PersistentObject *getUserData(ITEM_t key) const
- void removeUserData(ITEM_t key)

Mit Hilfe dieser Funktionen kann der Benutzer ein beliebiges Objekt, das ebenfalls von IR_PersistentObject abgeleitet ist, an eine beliebige Klasse der ICD-C-Datenstruktur anheften. Das angeheftete Objekt wird dabei durch einen Schlüssel (key) eindeutig identifiziert.

Wie beim Callback-Mechanismus ermöglicht auch diese Vorgehensweise ein Erweitern der ICD-C, ohne die ICD-C selbst verändern zu müssen.

3.1.1.4 ICD-C-Optimierungen

Dieser Abschnitt widmet sich den zahlreichen Compiler-Optimierungen der ICD-C, die spätestens auf der höchsten Optimierungsstufe des WCCs (O3) automatisch ausgeführt werden.

- **Merge String Constant Expressions**
  Mehrere konstante Zeichenketten, die sich nicht voneinander unterscheiden, werden zu einer Zeichenkette zusammengefasst.
- **Loop Collapsing**
  Verschachtelte Schleifen, die über ein mehrdimensionales Array iterieren, werden zu einer einzigen Schleife zusammengefasst. Die neue Schleife behandelt das mehrdimensionale Array wie ein eindimensionales Array.
- **Loop De-Indexing**
  Innerhalb von Schleifen werden Array-Zugriffe mittels des Index-Operators durch eine äquivalente Pointer-Arithmetik ersetzt.
- **Tail Recursion Elimination**
  Eine rekursive Funktion, die nur in der letzten Anweisung einen rekursiven Funktionsaufruf beinhaltet, wird in eine nicht-rekursive Funktion transformiert, indem der rekursive Funktionsaufruf durch einen Sprung zum Anfang der Funktion ersetzt wird.
- **Function Inlining**
  Ein Funktionsaufruf wird durch eine Kopie des Funktionsrumpfes der aufgerufenen Funktion ersetzt.
- **Remove Unused Symbols**
  Nicht gebrauchte Symbole werden aus dem Quellcode entfernt. Nicht gebrauchte Funktionen werden ebenfalls entfernt.
- **Expression Simplification**
  Ausdrücke werden mit Hilfe von Ersetzungsregeln vereinfacht. So werden z.B. Ausdrücke mit redundanten Casts oder sich gegenseitig aufhebende Operatoren wie der Adress-Operator "`&`" und der Dereferenzierungs-Operator "`*`" vereinfacht.
- **Fold Constant Code**
  Ausdrücke, die nur Konstanten beinhalten, werden schon zur Laufzeit des Compilers ausgewertet und durch das Ergebnis ersetzt. Außerdem werden Variablen, die als konstant deklariert wurden, durch ihren Wert ersetzt.

  Schleifen, `if`-Anweisungen und `switch`-Anweisungen, deren Bedingungen nur aus Konstanten bestehen, können weiter vereinfacht werden. So können z.B. `while`-Schleifen mit konstant falscher Bedingung entfernt werden.
- **Value Propagation**
  Konstante Integer-Ausdrücke werden nach Möglichkeit bis zu ihren Einsatzorten propagiert.
- **Dead Code Elimination**
  Anweisungen, die wirkungslos oder im Kontrollflussgraph unerreichbar sind, werden entfernt.

Zuweisungen, deren Ergebnis nie verwendet wird, werden ebenfalls entfernt. Außerdem werden, wenn möglich, direkt aufeinanderfolgende Basis-Blöcke (der Vorgänger hat genau einen Nachfolger, der Nachfolger genau einen Vorgänger) zu einem Basis-Block verschmolzen.

- **Eliminate Return Value**
  Falls der Rückgabewert einer Funktion bei keinem einzigen Aufruf der Funktion benutzt wird, wird in der Funktion auf die Rückgabe eines Funktionswertes verzichtet.
- **Procedure Cloning (Function Specialization)**
  Von bestimmten Funktionen, die mit konstanten Argumenten augerufen werden, werden spezialisierte Kopien (Clones) erzeugt. Diese müssen weniger Parameter als die ursprüngliche Funktion verwalten, weil bestimmte Parameter durch Konstanten ersetzt wurden.
- **Loop Unrolling**
  Schleifen werden um einen statisch ermittelten Abroll-Faktor u abgerollt, um die Anzahl der Sprünge zum Schleifenkopf zu reduzieren. Dafür muss der Schleifenrumpf $u - 1$ mal kopiert werden und das Verändern der Laufvariable muss entsprechend angepasst werden.
- **Loop Unswitching**
  Falls in einer Schleife eine Verzweigung enthalten ist, die zudem unabhängig von der umgebenden Schleife ist, wird diese Anweisung vor die Schleife verschoben.
- **Polyhedral If-Statement Optimization**
  Die Bedingungen von `if`-Anweisungen in verschachtelten Schleifen werden mit Hilfe von mathematischen Modellen optimiert.
- **Transform Head Controlled Loops**
  Kopfgesteuerte `for`- und `while`-Schleifen werden in eine fußgesteuerte `do-while`-Schleife transformiert, um die Anzahl der Sprünge zu reduzieren.
- **Struct Scalarization**
  Strukturen mit Bitfeldern werden in äquivalente Objekte vom Typ Integer transformiert. Außerdem werden skalare Komponenten einer Struktur in eigenständige Komponenten überführt.
- **Common Subexpression Elimination**
  Wiederholte Berechnungen von Teilausdrücken, die sich an unterschiedlichen Stellen im Quellcode befinden, werden erkannt, und durch das Ergebnis ersetzt, sodass eine wiederholte Berechnung vermieden wird.
- **Separate Life Ranges**
  Lokale Variablen, die in verschiedenen Lebensbereichen für unterschiedliche Zwecke eingesetzt werden, werden in mehrere einzelne Variablen aufgeteilt.
- **Redundant Load Elimination**
  Lesezugriffe auf globale Variablen innerhalb von Funktion werden vermieden, indem eine neue lokale Variable eingeführt und mit dem Wert der globalen Variable initialisiert wird.
- **Remove Unused Function Arguments**
  Nicht gebrauchte Funktionsparameter werden entfernt.

- **Create Multiple Exits**
  Falls unterschiedliche Verzweigungen im Kontrollfluss einer Funktion in dieselbe return-Anweisung münden, erhält jede Verzweigung eine eigene return-Anweisung, damit die Anzahl der Sprünge reduziert wird.

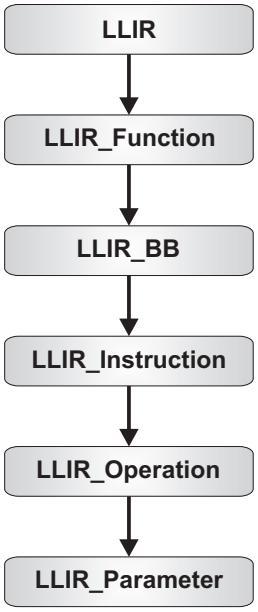


Abbildung 3.4: *ICD-LLIR-Aufbau*

3.1.2 ICD-LLIR

Die Low-Level IR ICD-LLIR (kurz LLIR) ist eine weitere Zwischendarstellung, die vom Informatik Centrum Dortmund entwickelt wurde. Im Vergleich zu ICD-C befindet sich die ICD-LLIR auf einer anderen Abstraktionsebene, da die ICD-LLIR eine Repräsentation von Maschinen-Code darstellt. Aus diesem Grund ist eine Transformation der ICD-LLIR in Assembler-Code einfach.

Bei der Entwicklung der ICD-LLIR wurde auf *Retargierbarkeit* geachtet, d.h. die ICD-LLIR kann auf unterschiedliche Prozessoren (z.B. DSPs, VLIWs, NPUs, ...) ohne großen Aufwand angepasst werden. Ein sehr generisch gehaltener Aufbau schafft die Voraussetzung für eine solche Flexibilität.

3.1.2.1 ICD-LLIR-Aufbau

Abb. 3.4 skizziert den Aufbau der ICD-LLIR.

Eine Instanz der Klasse LLIR entspricht einer Assembler-Datei. Die LLIR besteht aus Funktionen (LLIR_Function), die wiederum aus LLIR-Basis-Blöcken (LLIR_BB) bestehen. LLIR-Basis-Blöcke verwalten Instruktionen (LLIR_Instruction) in der Reihenfolge ihrer Abarbeitung und werden nur durch die erste Instruktion betreten und die letzte Instruktion verlassen. Eine Instruktion kann aus einem

oder mehreren Maschinen-Operationen (LLIR_Operation) bestehen. An dieser Stelle wird auch der generische Aufbau der ICD-LLIR sichtbar. So kann z.B. eine VLIW-Architektur, die mehrere Maschinen-Operationen parallel ausführen kann, problemlos modelliert werden. Eine Maschinen-Operationen wiederum enthält eine Assembler-Mnemonik (z.B. ADD, MUL, DIV, ...) und beliebig viele Parameter (LLIR_Parameter). Parameter können z.B. Register oder Integer-Konstanten repräsentieren. Der Aufbau der ICD-LLIR ist in [ICD-LLIR 2008] detailliert beschrieben.

3.1.2.2 ICD-LLIR-Analysen

Auch die ICD-LLIR bietet diverse Kontrollfluss- und Datenflussanalysen. Eine wichtige Datenflussanalyse, die im Rahmen dieser Diplomarbeit eingesetzt wird, ist die Lebendigkeitsanalyse (Life Time Analysis, LTA). Sie untersucht die Lebenszeiten von virtuellen Registern und wird in Abschnitt 3.1.5 näher erläutert.

Eine weitere Datenflussanalyse ist die DEF/USE-Analyse. Sie erzeugt einen DEF/USE-Abhängigkeitsgraphen für Register, aus dem die Schreib- und Lesezugriffe auf Register entnommen werden können. Die Lebendigkeitsanalyse basiert im wesentlichen auf DEF/USE-Informationen.

Darüber hinaus ermöglicht die ICD-LLIR wie die ICD-C ein effizientes Durchlaufen und Manipulieren der ICD-LLIR-Datenstruktur.

3.1.2.3 ICD-LLIR-Erweiterbarkeit

Auch die ICD-LLIR-Datenstruktur kann um beliebige Benutzerdaten erweitert werden, ohne die ICD-LLIR selbst verändern zu müssen. Für diesen Zweck gibt es sogenannte Objectives (LLIR_Objective), die über die Klasse LLIR_Handler verwaltet werden. Folgende Funktionen des LLIR_Handlers ermöglichen die Verwaltung von Objectives:

- void addObjective(LLIR_Objective *)
- LLIR_Objective *getObjective(ObjectiveType) const
- void removeObjective(ObjectiveType)

Mit Hilfe dieser Funktionen kann der Benutzer ein beliebiges Objekt, das von LLIR_Objective abgeleitet ist, an eine beliebige Klasse der ICD-LLIR-Datenstruktur anheften. Das angeheftete Objekt wird dabei durch einen Schlüssel (ObjectiveType) eindeutig identifiziert.

3.1.2.4 LLIR-Optimierungen

Dieser Abschnitt beschäftigt sich mit den Compiler-Optimierungen für die LLIR, die in bestimmten WCC-Optimierungsstufen automatisch aufgerufen werden. Die LLIR-Optimierungen des WCCs

werden unterteilt in virtuelle (prozessorunabhängige) und physikalische (prozessorabhängige) Optimierungen. Zuerst werden die virtuellen Optimierungen betrachtet, weil sie vor den physikalischen Optimierungen ausgeführt werden. Anschließend werden physikalische Optimierungen für den Infineon TriCore-Prozessor vorgestellt, die im WCC enthalten sind.

Virtuelle LLIR-Optimierungen der ICD-LLIR

Die ICD-LLIR stellt aufgrund ihrer generischen Beschaffenheit nur virtuelle Optimierungen bereit, die unabhängig von einem bestimmten Prozessor sind.

- **Loop Invariant Code Motion**
  Falls eine Instruktion unabhängig von der umgebenden Schleife ist, wird sie vor die Schleife verschoben.
- **Remove Empty Basic Blocks**
  LLIR-Basis-Blöcke, die keine Instruktionen beinhalten oder im Kontrollflussgraph unerreichbar sind, werden entfernt.
- **Remove Unused Virtual Registers**
  Nicht gebrauchte virtuelle Register werden entfernt.

Virtuelle LLIR-Optimierungen des WCCs

- **Dead Code Elimination**
  Operationen, die nachweislich keinen Effekt auf das Ergebnis einer Berechnung haben, werden entfernt.
- **Redundant Code Detection**
  Wiederholte Berechnungen auf Bit-Ebene werden erkannt. Anschließend wird das Ergebnis zwischengespeichert, sodass eine erneute Berechnung vermieden wird.
- **Fold Constant Code**
  Konstanten auf Bit-Ebene werden schon zur Laufzeit des Compilers ausgewertet und durch das Ergebnis ersetzt.
- **Constant Propagation**
  Konstanten auf Bit-Ebene werden nach Möglichkeit bis zu ihren Einsatzorten propagiert.
- **Peephole Optimization**
  Die Assembler-Befehle werden nach bestimmten Mustern durchsucht und ggf. durch eine gewinnversprechende Folge von Assembler-Befehlen ersetzt. Außerdem werden redundante Befehle entfernt.
- **Merge Redundant Basic Blocks**
  Direkt aufeinanderfolgende LLIR-Basis-Blöcke (der Vorgänger hat genau einen Nachfolger, der

Nachfolger genau einen Vorgänger, und die letzte Instruktion des Vorgängers ist *keine* Sprung-Instruktion) werden zu einem LLIR-Basis-Block verschmolzen.

Physikalische LLIR-Optimierungen des WCCs

- **Generate 16-bit Instructions**
  32-bit Instruktionen werden, soweit möglich, durch äquivalente 16-bit Instruktionen ersetzt.
- **Adjust Jump Displacements**
  Sprung-Instruktionen mit zu großen Sprungzielen, die durch zuvor ausgeführte Optimierungen entstanden sind, werden angepasst, um die Korrektheit des Codes zu gewährleisten.
- **Correct Instruction Types**
  Die Instruktions-Typen müssen nach der Register-Allokation auf Korrektheit überprüft und ggf. modifiziert werden. So werden z.B. Instruktionen mit zu großen Offset-Werten angepasst, indem die Offset-Werte durch Register-Operanden ersetzt werden.

3.1.3 Flow Fact-Manager

Die statische WCET-Analyse gehört zu den unentscheidbaren Problemen dieser Welt. Aus diesem Grund sind zusätzliche Kontrollflussinformationen, die im Folgenden **Flow Facts** genannt werden, für eine WCET-Analyse zwingend notwendig. Sie geben Auskunft über maximale Iterationsanzahl von Schleifen und maximale Rekursionstiefen.

Flow Facts werden durch den Benutzer mit Hilfe von Pragma-Direktiven direkt im Quellcode annotiert. Das folgende Beispiel zeigt eine annotierte `for`-Schleife:

```
1 int i;
2 int sum = 0;
3
4 _Pragma( "loopbound min 10 max 10" )
5 for ( i = 1; i <= 10; ++i ) {
6   sum += i;
7 }
```

Abbildung 3.5: *Beispiel für eine annotierte `for`-Schleife*

Die Annotation in Zeile vier spezifiziert die minimale und maximale Iterationsanzahl der zugehörigen `for`-Schleife. Die Schleife wird in diesem Beispiel (jedes mal) genau zehn mal iteriert (min = max = 10). Die Iterationsanzahl der Schleife ist somit nicht *variabel*, sondern *konstant*.

Im Laufe der Transformationen durch Optimierungen wie z.B. Loop Unrolling können Flow Facts ungültig werden. In einem solchen Fall müssen die Flow Facts angepasst werden. Das ist die Aufgabe des Flow Fact-Managers. Der Flow Fact-Manager ist zuständig für eine automatische Flow Fact-

Konsistenzsicherung. Die Modellierung und Transformation von Flow Facts wird in [Schulte 2007] detailliert erklärt.

Flow Facts sind für die vorliegende Arbeit von Bedeutung, weil der ICD-C Loop Analyzer nicht jede Schleife analysieren kann. Trotzdem kommt der ICD-C Loop Analyzer in einigen ICD-C-Optimierungen wie z.B. Loop Unrolling zum Einsatz. In dieser Arbeit werden Flow Facts ausgenutzt, um mehr Potential für die neuartige WCET-fähige Optimierung Loop Unrolling zu schaffen (siehe Kapitel 7).

3.1.4 Code-Selektor

Der Code-Selektor gehört zu den zwingend notwendigen Stationen des Compilers und wird auch das "Herz" des Compilers genannt, weil hier die eigentliche Transformation der Quell- in Zielsprache stattfindet. Im Code-Selektor wird die (optimierte) High-Level IR ICD-C in die Low-Level IR ICD-LLIR überführt. Die ICD-LLIR-Datenstruktur (LLIR, LLIR_Function, LLIR_BB, ...) wird folglich im Code-Selektor aufgebaut.

Der Code-Selektor des WCCs erzeugt bei der Instruktionsauswahl eine *virtuelle* LLIR. Die virtuelle LLIR darf eine beliebige Anzahl virtueller Register verwenden. Natürlich hat der Prozessor nur eine begrenzte Anzahl an physikalischen Registern. Deswegen muss in einem weiteren Schritt die virtuelle LLIR in eine physikalische LLIR transformiert werden. Dies ist die Aufgabe der Register-Allokation.

Der Code-Selektor ist für die vorliegende Diplomarbeit deswegen so interessant, weil die meisten LLIR-Basis-Blöcke im Code-Selektor erzeugt werden. Für eine Back-annotation von detaillierten WCET_{est}-Informationen muss im Code-Selektor eine Verbindung zwischen den IR- und LLIR-Basis-Blöcken hergestellt werden, weil beim Erzeugen von neuen LLIR-Basis-Blöcken die zugehörigen IR-Basis-Blöcke leicht erkennbar sind. Diese Verbindung ist notwendig für eine Transformation von WCET_{est}-Informationen von der Low-Level IR ICD-LLIR in die High-Level IR ICD-C. In den späteren Phasen des Compilers ist die Verbindung zwischen IR- und LLIR-Basis-Blöcken nicht mehr ersichtlich.

3.1.5 Register-Allokation

Die Register-Allokation beschreibt den Vorgang der Abbildung virtueller auf physikalische Register. Ziel der Register-Allokation ist die bestmögliche Ausnutzung der knappen Ressource von Prozessor-Registern. Damit zwei virtuelle Register auf das gleiche physikalische Register abgebildet werden können, muss u.a. sichergestellt werden, dass sich die Lebenszeiten der virtuellen Register nicht überlappen. Dabei wird ein virtuelles Register als lebendig angesehen, wenn es einen Wert enthält, der in Zukunft noch gebraucht werden *könnte*. Die Lebendigkeitsanalyse untersucht, wann die Lebenszeiten von virtuellen Registern beginnen und enden.

Falls bei der Register-Allokation kein physikalisches Register mehr frei ist, entsteht Spilling. Spilling bezeichnet den Prozess des Aus- und Einlagerns von Registerinhalten in den und aus dem langsameren Speicher. Spill-Code hat natürlich negative Auswirkungen auf die Programmlaufzeit, weil Zugriffe

auf einen langsameren Speicher im Vergleich zu den effizienten Register-Zugriffen sehr hohe Kosten verursachen. Aus diesem Grund sollten High-Level Compiler-Optimierungen die Register-Allokation nicht unberücksichtigt lassen. Kapitel 8 wird zeigen, wie wichtig die Ergebnisse der Lebendigkeitsanalyse für das Lernen einer Heuristik für die Optimierung Function Inlining sind.

3.1.6 Back-annotation

WCET$_{est}$-Informationen sind zu Beginn der Diplomarbeit in der ICD-C nur auf Funktions- und Übersetzungseinheit-Ebene verfügbar. Detaillierte WCET$_{est}$-Informationen, die in der ICD-LLIR auf Basis-Block-Ebene verfügbar sind, sind in der ICD-C nicht vorhanden. Deswegen können High-Level Compiler-Optimierung, die eine WCET-Minimierung zum Ziel haben, nicht von diesen Informationen profitieren.

Eine notwendige Voraussetzung für die Transformation von detaillierten WCET$_{est}$-Informationen in die ICD-C ist die Herstellung einer Verbindung zwischen ICD-C und ICD-LLIR auf Basis-Block-Ebene. Im nächsten Kapitel wird daher erklärt, wie und wo eine Verbindung zwischen IR- und LLIR-Basis-Blöcken hergestellt werden kann. Anschließend wird auf die Transformation von detaillierten WCET$_{est}$-Informationen eingegangen.

Nach der Transformation stehen detaillierte WCET$_{est}$-Informationen auf High-Level IR-Ebene zur Verfügung. In Kapitel 5 wird gezeigt, wie man aus den transformierten WCET$_{est}$-Informationen Cold Path-Informationen gewinnen kann. Der Cold Path demonstriert, wie man WCET$_{est}$-Informationen ausnutzen kann, um z.B. die Anzahl der WCET-Analysen zu reduzieren.

Wie man andererseits die WCET$_{est}$- und Cold Path-Informationen für die bereits bestehenden High-Level Compiler-Optimierungen Function Inlining und Loop Unrolling ausnutzen kann, um eine WCET-Minimierung zu erreichen, wird in den Kapiteln 6 und 7 beschrieben.

4 Back-annotation

Dieses Kapitel behandelt das Thema Back-annotation: Transformation von WCET$_{est}$-Informationen von der Low-Level IR ICD-LLIR in die High-Level IR ICD-C. WCET$_{est}$-Informationen von Funktionen und Übersetzungseinheiten stehen bereits zu Beginn der Diplomarbeit in der ICD-C zur Verfügung. Ein Ziel dieser Diplomarbeit ist die Transformation von detaillierten WCET$_{est}$-Informationen, die in der ICD-LLIR auf Basis-Block-Ebene verfügbar sind, in die ICD-C.

4.1 Motivation

Optimierungen, die eine WCET-Minimierung zum Ziel haben, müssen auf dem WCEP ausgeführt werden, damit sich die WCET verbessert. WCET-Optimierungen, die nicht auf dem WCEP ausgeführt werden, führen in der Regel nicht zu einer Verbesserung der WCET. Ein großes Problem von WCET-Optimierungen besteht darin, dass der WCEP im Gegensatz zum ACEP im Laufe der Transformationen durch Optimierungen wechseln kann. Dieser Umstand wird im nächsten Kapitel anhand eines Beispiels ausführlich erläutert. WCET-Optimierungen müssen damit rechnen, dass der WCEP nach jedem Optimierungsschritt wechselt. Folglich *müssen* High-Level Compiler-Optimierungen, die eine WCET-Minimierung zum Ziel haben, auf (detaillierte) WCET$_{est}$-Informationen zugreifen, um nach jedem Optimierungsschritt auf dem gültigen WCEP optimieren zu können. Dies stellt die Hauptmotivation für dieses Kapitel dar. Darüber hinaus können High-Level Compiler-Optimierungen unter Ausnutzung von detaillierten WCET$_{est}$-Informationen Teile des Programms ermitteln, die eine relativ hohe WCET$_{est}$ verursachen, und entsprechend darauf reagieren.

Detaillierte WCET$_{est}$-Informationen werden nach der WCET-Analyse in Objectives vom Typ `WCET` gespeichert (vgl. Abschnitt 3.1.2.3). Die Objectives werden anschließend an LLIR-Basis-Blöcke angeheftet. Die nachstehenden Informationen, die im `WCET`-Objective eines jeden LLIR-Basis-Blocks gespeichert sind, sollen in die ICD-C transformiert werden:

- **WCET**$_{est}$
  Die WCET$_{est}$, die durch einen LLIR-Basis-Block im gesamten Programmlauf verursacht wird. Der Wert berechnet sich aus der Summe der WCET$_{est}$-Werte der einzelnen Kontexte des LLIR-Basis-Blocks. Somit entspricht die globale WCET$_{est}$ des zu analysierenden Programms P der Summe der WCET$_{est}$ aller LLIR-Basis-Blöcke BB:

$$WCET_{est}(P) = \sum_{BB} WCET_{est}(BB) \tag{4.1}$$

- **WCEP**
  Ein `WCET`-Objective gibt auch Auskunft darüber, ob der betrachtete LLIR-Basis-Block auf dem WCEP liegt oder nicht.
- **WCEC**$_{sum}$ (Summarized Worst-Case Execution Count)
  Die Anzahl der Ausführungen eines LLIR-Basis-Blocks, die sich aus der Summe der Ausführungen des LLIR-Basis-Blocks über alle Kontexte hinweg ergibt.
- **Infeasible**
  Dieses Attribut kennzeichnet LLIR-Basis-Blöcke, die in allen Kontexten unausführbar sind.

Im folgenden Abschnitt wird zuerst die bereits vorhandene Transformation für Funktionen und Übersetzungseinheiten beschrieben. Anschließend wird die Transformation von detaillierten WCET$_{est}$-Informationen vorgestellt, die im Rahmen dieser Diplomarbeit entwickelt wurde. Probleme werden erläutert und Lösungsansätze werden präsentiert. Außerdem wird eine kurze Beschreibung der Datenstrukturen und Schnittstellen geboten. Abschnitt 4.3 geht auf die Integration des Back-annotation-Moduls in den WCC ein. Abschnitt 4.4 behandelt schließlich die Verifikation der transformierten WCET$_{est}$-Informationen.

4.2 Transformation von WCET$_{est}$-Informationen von der ICD-LLIR in die ICD-C

Damit WCET$_{est}$-Informationen von der ICD-LLIR in die ICD-C transformiert werden können, muss zuerst eine Verbindung zwischen diesen beiden Zwischendarstellungen hergestellt werden. Die Herstellung der Verbindung für Übersetzungseinheiten gestaltet sich einfach, weil der Code-Selektor des WCCs für jede C-Datei, die in der ICD-C durch eine Instanz der Klasse IR_CompilationUnit repräsentiert wird, genau eine Instanz der Klasse LLIR erzeugt. Folglich existiert eine 1-zu-1-Beziehung zwischen IR- und LLIR-Übersetzungseinheiten. In Abb. 4.1 wird veranschaulicht, wie die Transformation von WCET$_{est}$-Informationen auf der Ebene der Übersetzungseinheiten durchgeführt wird.

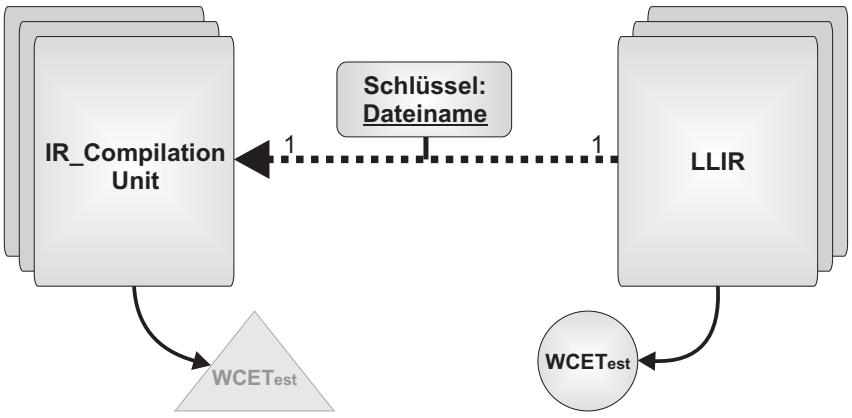


Abbildung 4.1: *Anheften von WCET$_{est}$-Informationen an IR-Übersetzungseinheiten*

Nach der aiT WCET-Analyse wird für jede LLIR-Übersetzungseinheit ein `WCET`-Objective erzeugt.

In diesem Objective werden die von aiT ermittelten WCET$_{est}$-Informationen für die gesamte LLIR-Übersetzungseinheit gespeichert. Danach kann die Back-annotation durchgeführt werden. Das Back-annotation-Modul kann über den Dateinamen die zugehörige IR-Übersetzungseinheit einer jeden LLIR-Übersetzungseinheit ermitteln. Anschließend erfolgt die Transformation von WCET$_{est}$-Informationen. Für jede IR-Übersetzungseinheit wird ein IR_PersistentObject erzeugt, das über den Schlüssel `WCET` eindeutig identifiziert werden kann (vgl. Abschnitt 3.1.1.3). In diese Datenstruktur werden die transformierten WCET$_{est}$-Informationen gespeichert.

Auch die Transformation von WCET$_{est}$-Informationen zwischen IR- und LLIR-Funktionen gestaltet sich einfach, weil wiederum eine eindeutige Verbindung zwischen Instanzen der Klasse IR_Function und LLIR_Function gegeben ist. In Abb. 4.2 wird ersichtlich, dass jede IR-Funktion genau eine zugehörige LLIR-Funktion besitzt und umgekehrt. Die Verbindung wird über den Funktionsnamen hergestellt und die Transformation wird analog zu der Transformation für Übersetzungseinheiten durchgeführt.

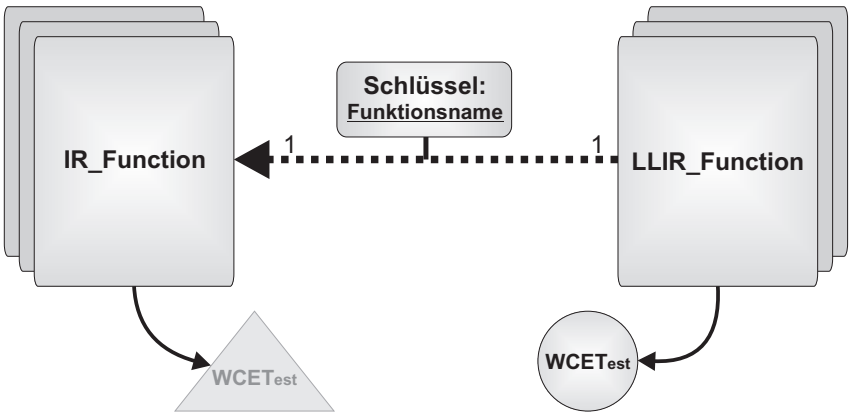


Abbildung 4.2: *Anheften von WCET$_{est}$-Informationen an IR-Funktionen*

Bei Funktionen, die als `static` deklariert sind, muss beachtet werden, dass diese nur innerhalb der Übersetzungseinheit, in der sie definiert wurden, sichtbar sind. Daher können zwei `static`-Funktionen, die in unterschiedlichen Übersetzungseinheiten eines Programms definiert sind, den gleichen Namen besitzen. Aus diesem Grund wird bei `static`-Funktionen neben dem Funktionsnamen auch der Dateiname der zugehörigen Übersetzungseinheit berücksichtigt, um weiterhin eine eindeutige Abbildung gewährleisten zu können.

Im Folgenden wird die Transformation von detaillierten WCET$_{est}$-Informationen vorgestellt, die in der ICD-LLIR auf Basis-Block-Ebene verfügbar sind. Die Transformation wurde im Rahmen dieser Diplomarbeit entwickelt und ist im Gegensatz zu den beiden vorgestellten Transformationen nicht trivial, da es keine eindeutige Verbindung zwischen IR- und LLIR-Basis-Blöcken gibt. Es besteht eine m-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken. Die Problematik wird in Abb. 4.3 auf der nächsten Seite dargestellt.

Nachfolgend wird anhand von Beispielen erläutert, warum es eine m-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken gibt und wie trotz der m-zu-n-Beziehung eine Verbindung zwischen IR- und LLIR-

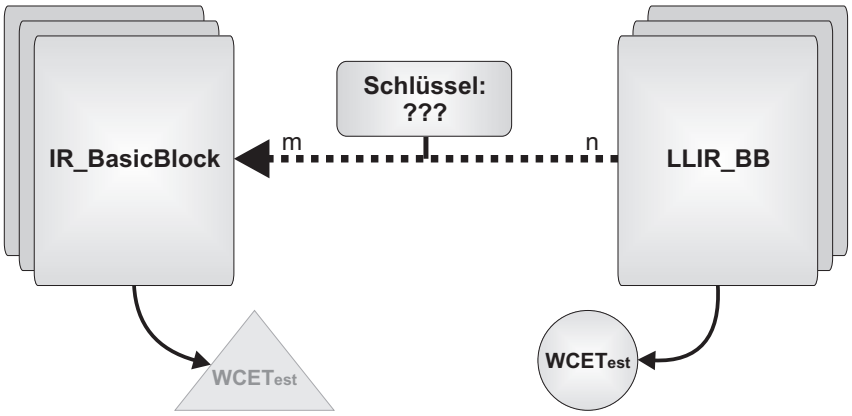


Abbildung 4.3: *Anheften von WCET$_{est}$-Informationen an IR-Basis-Blöcke*

Basis-Blöcken hergestellt werden kann. Anschließend wird auf die Transformation von detaillierten WCET$_{est}$-Informationen näher eingegangen.

4.2.1 Herstellung der Verbindung zwischen IR- und LLIR-Basis-Blöcken

Die Herstellung der Verbindung zwischen IR- und LLIR-Basis-Blöcken beginnt mit der Instruktionsauswahl durch den Code-Selektor (vgl. Abschnitt 3.1.4).

Die 1-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken

Meistens gibt es zwischen IR- und LLIR-Basis-Blöcken eine 1-zu-1-Beziehung (vgl. Abb. 4.4).

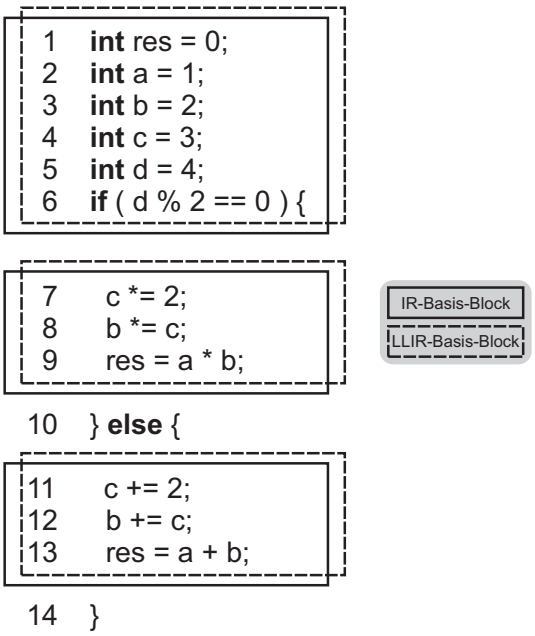


Abbildung 4.4: *Beispiel für eine 1-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken*

In Abb. 4.4 ist ein Ausschnitt aus einem C-Quellcode zu sehen. Der Quellcode enthält eine `if-else`-Verzweigung und wird in der ICD-C-Repräsentation durch drei IR-Basis-Blöcke *überdeckt*. Im Folgenden überdeckt ein IR- bzw. LLIR-Basis-Block einen Ausschnitt aus einem C-Quellcode, wenn alle zugehörigen Anweisungen bzw. Instruktionen durch diesen Basis-Block repräsentiert werden.

Im Beispiel aus Abb. 4.4 wird der sequentielle Code in den Zeilen 1 bis 6, der auch die Bedingung der `if-else`-Verzweigung enthält, durch einen IR-Basis-Block überdeckt. Zudem überdeckt jeweils ein IR-Basis-Block den `then`- und `else`-Teil der `if-else`-Verzweigung. In Abb. 4.4 ist auch ersichtlich, welche Code-Anteile in der zugehörigen ICD-LLIR-Repräsentation durch LLIR-Basis-Blöcke überdeckt werden. Es wird deutlich, dass es eine 1-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken gibt. Für die Verwaltung der 1-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken wird ein Wörterbuch eingeführt:

Wörterbuch LLIR- nach IR-Basis-Block: String $\longrightarrow$ *IR-Basis-Block*

Dieses Wörterbuch ermöglicht die Abbildung von LLIR- auf IR-Basis-Blöcke und ist ein wichtiger Schritt zur Herstellung der notwendigen Verbindung. Als Schlüssel dient ein Name vom Typ String, der sich aus dem Namen der zugehörigen LLIR eines LLIR-Basis-Blocks und dessen Label-Namen zusammensetzt, um Mehrdeutigkeiten zu vermeiden. In Abschnitt 3.1.4 wurde erwähnt, dass die meisten LLIR-Basis-Blöcke im Code-Selektor erzeugt werden. Daher wird das Wörterbuch größtenteils während der Instruktionsauswahl durch den Code-Selektor gefüllt.

Die 1-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken

Die 1-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken wird in erster Linie durch die unterschiedlichen Definitionen des Begriffs Basis-Block verursacht. In der ICD-LLIR verwalten LLIR-Basis-Blöcke Instruktionen in der Reihenfolge ihrer Abarbeitung und werden nur durch die erste Instruktion betreten und die letzte Instruktion verlassen. IR-Basis-Blöcke der ICD-C hingegen verwalten Anweisungen in der Reihenfolge ihrer Abarbeitung und dürfen zusätzlich über einen Funktionsaufruf *innerhalb* des Basis-Blocks verlassen und anschließend wieder betreten werden. Dies wird in Abb. 4.5 verdeutlicht.

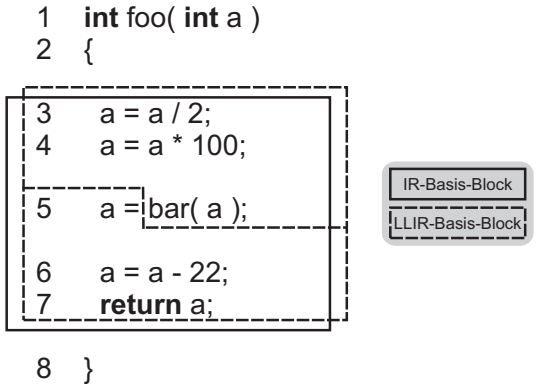


Abbildung 4.5: *Beispiel für eine 1-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken*

Der gesamte Funktionsrumpf wird durch einen IR-Basis-Block überdeckt. Für diesen IR-Basis-Block wird im Code-Selektor zunächst ein LLIR-Basis-Block erzeugt. Der Funktionsaufruf `bar( a )` in Zeile 5 führt dazu, dass der zugehörige LLIR-Basis-Block im Code-Selektor an der Stelle des Funktionsaufrufs in zwei LLIR-Basis-Blöcke aufgeteilt wird. Die Instruktionen nach dem Funktionsaufruf werden in den neu erzeugten LLIR-Basis-Block verschoben.

Im Wörterbuch werden die beiden LLIR-Basis-Blöcke dem gleichen IR-Basis-Block zugeordnet. Deshalb ist die Abbildung, die durch das Wörterbuch beschrieben wird, nicht injektiv.

Einen weiteren Grund für die 1-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken stellen komplexe Bedingungen mit mehr als einem Vergleich dar. Abb. 4.6 zeigt ein Beispiel für eine komplexe Bedingung einer `if`-Verzweigung.

```
1 if ( ( a == 1 ) && ( b == 2 ) || ( c == 3 ) ) {
2   /* then-Block */
3 }
```

Abbildung 4.6: *Beispiel für eine komplexe Bedingung*

Die Bedingung in Zeile 1 besteht dabei aus drei Vergleichen. Alle Vergleiche werden durch einen IR-Basis-Block überdeckt, wohingegen in der ICD-LLIR jeder Vergleich durch einen eigenen LLIR-Basis-Block überdeckt wird. Auch in diesem Fall werden für alle LLIR-Basis-Blöcke neue Einträge im Wörterbuch erzeugt, die alle auf den gleichen IR-Basis-Block verweisen.

Die 1-zu-0-Beziehung zwischen IR- und LLIR-Basis-Blöcken

Nach der Instruktionsauswahl durch den Code-Selektor werden die LLIR-Optimierungen ausgeführt. Ähnlich wie bei den Flow Facts müssen die Datenstrukturen des Back-annotation-Moduls im Laufe der LLIR-Optimierungen gegebenenfalls angepasst werden, um eine gültige Abbildung zu gewährleisten. Die LLIR-Optimierung *Remove Empty Basic Blocks* z.B. entfernt LLIR-Basis-Blöcke, die aufgrund vorangegangener LLIR-Optimierungen keine Instruktionen beinhalten oder im Kontrollflussgraph unerreichbar sind. Der zugehörige IR-Basis-Block eines LLIR-Basis-Blocks wird aber nicht entfernt, weil die ICD-C nach der Instruktionsauswahl durch den Code-Selektor nicht weiter optimiert wird. IR-Basis-Blöcke, deren zugehörige LLIR-Basis-Blöcke entfernt wurden, werden im Back-annotation-Modul in einer Liste verwaltet.

Die m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken

Das Beispiel in Abb. 4.7 demonstriert, dass es auch eine m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken geben kann.

In Abb. 4.7 ist eine `do-while`-Schleife abgebildet. Schleifenrumpf und Schleifenbedingung werden jeweils durch einen IR-Basis-Block überdeckt. Der Code-Selektor erzeugt bei der Instruktionsauswahl

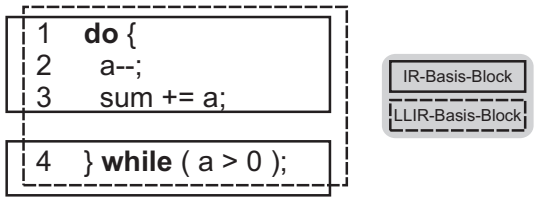


Abbildung 4.7: *Beispiel für eine m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken*

für jeden dieser IR-Basis-Blöcke einen zugehörigen LLIR-Basis-Block. Aufgrund der LLIR-Optimierung *Merge Redundant Basic Blocks* jedoch werden die beiden LLIR-Basis-Blöcke zu *einem* LLIR-Basis-Block verschmolzen: Ein LLIR-Basis-Block wird gelöscht und der andere LLIR-Basis-Block erhält die Instruktionen des gelöschte LLIR-Basis-Blocks.

Eine m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken muss dem Back-annotation-Modul bekanntgegeben werden, weil es ansonsten IR-Basis-Blöcke geben kann, die keine zugehörigen LLIR-Basis-Blöcke besitzen. Solche IR-Basis-Blöcke können in der Back-annotation-Phase aufgrund einer fehlenden Verbindung nicht annotiert werden. Um das zu verhindern, werden im Folgenden *Referenzen* auf IR-Basis-Blöcke vorgestellt.

Wenn es eine m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken gibt, müssen sich die IR-Basis-Blöcke die WCET$_{est}$-Informationen des zugehörigen LLIR-Basis-Blocks "teilen". Die WCET$_{est}$-Informationen des LLIR-Basis-Blocks dürfen nicht dupliziert werden, weil die mehrfache Betrachtung von WCET$_{est}$-Informationen eines LLIR-Basis-Blocks die globale WCET$_{est}$ auf High-Level IR-Ebene verfälscht. Deswegen werden die WCET$_{est}$-Informationen nur an *einem* frei wählbaren IR-Basis-Block angeheftet. Alle anderen IR-Basis-Blöcke, die WCET$_{est}$-Informationen mit diesem IR-Basis-Block teilen müssen, speichern eine Referenz auf diesen IR-Basis-Block. Das verhindert eine wiederholte Speicherung von WCET$_{est}$-Informationen in der ICD-C, die zu einer Verfälschung der globalen WCET$_{est}$ auf High-Level IR-Ebene führt. Für diesen Zweck wird ein neues Wörterbuch eingeführt:

Wörterbuch IR-Basis-Block Referenzen: IR-Basis-Block $\longrightarrow$ Liste von IR-Basis-Blöcken

Das Wörterbuch verwaltet für jeden IR-Basis-Block eine Liste von IR-Basis-Blöcken, die diesen IR-Basis-Block referenzieren. Wäre die Abbildung anders herum (IR-Basis-Block wird auf seinen Referenz IR-Basis-Block abgebildet), könnte es im Wörterbuch mehrere Einträge für einen Referenz IR-Basis-Block geben, weil ein Referenz IR-Basis-Block durch mehrere IR-Basis-Blöcke referenziert werden kann.

Zusammengefasst lässt sich sagen, dass die Verbindung zwischen IR- und LLIR-Basis-Blöcken im Code-Selektor aufgebaut werden muss. Diese Verbindung muss im Laufe der LLIR-Optimierungen konsistent gehalten werden. Die dafür notwendigen Funktionen werden im Folgenden kurz erläutert.

Schnittstellen

Die Herstellung der Verbindung zwischen IR- und LLIR-Basis-Blöcken beginnt mit der Instruktionsauswahl durch den Code-Selektor. Die im Code-Selektor erzeugten LLIR-Basis-Blöcke beinhalten noch keine WCET_{est}-Informationen, weil WCET_{est}-Informationen erst nach der WCET-Analyse verfügbar sind. Die WCET-Analyse wird, wie in Kapitel 3 beschrieben, frühestens *nach* den LLIR-Optimierungen aufgerufen. Im Code-Selektor wird hauptsächlich die Funktion `addNewMapping` benötigt:

- `void addNewMapping( LLIR_BB *llirBB, IR_BasicBlock *irBB )`
  Diese Funktion wird im Code-Selektor nach dem Erzeugen des entsprechenden LLIR-Basis-Blocks eines IR-Basis-Blocks aufgerufen, um eine 1-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken bekannt zu geben. Sie erstellt automatisch einen passenden Eintrag im Wörterbuch *LLIR- nach IR-Basis-Block* und wird *ausschließlich* im Code-Selektor aufgerufen, weil LLIR-Optimierungen auf die Schnittstellen der ICD-C (wie z.B. IR_BasicBlock) nicht zugreifen dürfen. Der Grund dafür wird im Folgenden erklärt.

Die durch den Code-Selektor aufgebaute ICD-LLIR wird anschließend durch LLIR-Optimierungen optimiert. Es wurde bereits erwähnt, dass die Verbindung zwischen IR- und LLIR-Basis-Blöcken, die im Code-Selektor hergestellt wurde, im Laufe der LLIR-Optimierungen konsistent gehalten werden muss. Da es sich bei der ICD-C und ICD-LLIR um eigenständige Module handelt, die getrennt voneinander in unterschiedlichen Projekten benutzt werden dürfen, musste darauf geachtet werden, dass die Schnittstellen zur Konsistenzsicherung keine Abhängigkeiten zwischen diesen Modulen erzeugen. Zudem dürfen keine Abhängigkeiten zwischen diesen Modulen und dem Back-annotation-Modul bestehen. Außerdem sollten `void`-Pointer aus Sicherheitsgründen vermieden werden.

Um den Anforderungen gerecht zu werden, wurde ein Konzept entwickelt, das den Callback-Mechanismus ausnutzt. Für diesen Zweck wurde ein neues Objective vom Typ `BACKANNOT` eingeführt. Um Abhängigkeiten zu vermeiden, rufen LLIR-Optimierungen die Funktionen zur Konsistenzsicherung über dieses Objective auf. Dieses Objective verwaltet Funktionspointer auf die Funktionen `addJoinMapping`, `addRefMapping` und `addDeadMapping`, welche im Back-annotation-Modul definiert werden. Diese Funktionen dienen zur Konsistenzsicherung der Verbindung zwischen IR- und LLIR-Basis-Blöcken und werden weiter unten näher erläutert.

Das Back-annotation-Modul erzeugt ein `BACKANNOT`-Objective für alle LLIR-Basis-Blöcke, für die `addNewMapping` im Code-Selektor aufgerufen wurde, und initialisiert anschließend die Funktionspointer im Objective. Ohne das Back-annotation-Modul werden auch keine `BACKANNOT`-Objectives erzeugt, sodass Abhängigkeiten vermieden werden. In Abb. 4.8 wird ersichtlich, dass dieses Objective als Schnittstelle zwischen dem Back-annotation-Modul und den LLIR-Optimierungen dient.

Im Folgenden werden die Funktionen `addJoinMapping`, `addRefMapping` und `addDeadMapping` kurz erläutert. Die Schnittstellen der Funktionen sind unabhängig von der ICD-C, sodass innerhalb der LLIR-Optimierungen über Funktionspointer auf diese Funktionen zugegriffen werden kann.

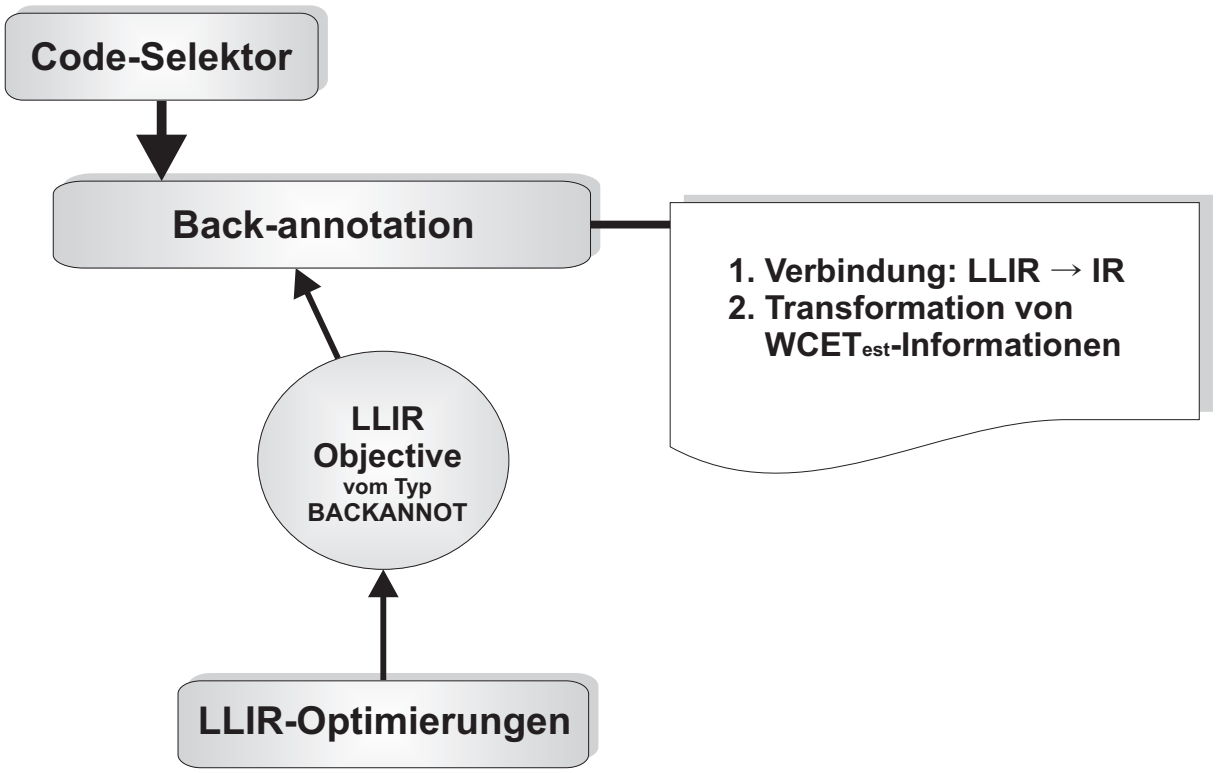


Abbildung 4.8: *LLIR Objective vom Typ* `BACKANNOT`

- `void addJoinMapping( LLIR_BB *bb, LLIR_BB *newFellow )`
  Mit Hilfe dieser Funktion wird eine 1-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken bekanntgegeben. Der neu erzeugte LLIR-Basis-Block `newFellow` wird auf den zugehörigen IR-Basis-Block von `bb` abgebildet. Diese Funktion ruft intern die Funktion `addNewMapping` auf, um einen neuen Eintrag im Wörterbuch *LLIR- nach IR-Basis-Block* zu erzeugen.
- `void addRefMapping( LLIR_BB *bb, LLIR_BB *target )`
  Anhand dieser Funktion wird eine m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken bekanntgegeben. Für diesen Zweck wird automatisch ein neuer Eintrag im Wörterbuch *IR-Basis-Block Referenzen* erstellt. In der Back-annotation-Phase wird der zugehörige IR-Basis-Block von `bb` eine Referenz auf den zugehörigen IR-Basis-Block von `target` speichern.
- `void addDeadMapping( LLIR_BB *bb )`
  Diese Funktion wiederum informiert das Back-annotation-Modul darüber, dass der LLIR-Basis-Block `bb` gelöscht wurde. IR-Basis-Blöcke, deren LLIR-Basis-Blöcke gelöscht wurden, werden in einer Liste verwaltet.

```
/* ---------- Aufruf der Funktion addJoinMapping ---------- */

// 'bb' ist ein LLIR-Basis-Block der Funktion f und
// soll einen neuen Nachfolger-Basis-Block 'bb2' erhalten
LLIR_BB *bb  = ...;

// 'bb2' wird neu erzeugt
LLIR_BB *bb2 = new LLIR_BB( getUniqueLabel().c_str() );

// 'bb2' wird ein Nachfolger-Basis-Block von 'bb'
f->InsertBB( bb2, bb );

// Back-annotation-Konsistenzsicherung
if ( bb->getHandler().hasObjective( LLIR_Handler::BACKANNOT ) ) {
  ( dynamic_cast< LLIR_BackAnnot * >(
     bb->getHandler().getObjective( LLIR_Handler::BACKANNOT ) ) )->
  callAddJoinMapping( bb, bb2 );
}
```

```
/* ---------- Aufruf der Funktion addRefMapping ---------- */

// 'bb' und 'bb2' sollen verschmolzen werden
LLIR_BB *bb  = ...;
LLIR_BB *bb2 = ...;

// Instruktionen von 'bb2' nach 'bb' verschieben
for ( LLIR_Instruction *i = bb2->GetFirstIns(); i;
      i = bb2->GetFirstIns() ) {
  bb2->MoveIns( i, bb->GetLastIns(), bb );
}

// Flow Facts von 'bb2' nach 'bb' verschieben
...

// Back-annotation-Konsistenzsicherung
if ( bb->getHandler().hasObjective( LLIR_Handler::BACKANNOT ) ) {
  ( dynamic_cast< LLIR_BackAnnot * >(
     bb->getHandler().getObjective( LLIR_Handler::BACKANNOT ) ) )->
  callAddRefMapping( bb2, bb );
}

delete( bb2 );
```

```
/* ---------- Aufruf der Funktion addDeadMapping ---------- */

// 'bb' soll geloescht werden
LLIR_BB *bb = ...;

// Back-annotation-Konsistenzsicherung
if ( bb->getHandler().hasObjective( LLIR_Handler::BACKANNOT ) ) {
  ( dynamic_cast< LLIR_BackAnnot * >(
     bb->getHandler().getObjective( LLIR_Handler::BACKANNOT ) ) )->
  callAddDeadMapping( bb );
}

delete( bb );
```

Abbildung 4.9: *Aufruf der Funktionen `addJoinMapping`, `addRefMapping` und `addDeadMapping`*

Um auch in Zukunft eine gültige Abbildung zwischen LLIR- und IR-Basis-Blöcken zu gewährleisten, müssen neue LLIR-Optimierungen über diese Funktionen das Erzeugen, Verschmelzen und Löschen von LLIR-Basis-Blöcken bekanntgeben. In Abb. 4.9 auf der vorherigen Seite wird anhand von Beispielen demonstriert, wie innerhalb der ICD-LLIR über das Objective vom Typ `BACKANNOT` die Funktionen `addJoinMapping`, `addRefMapping` und `addDeadMapping` aufgerufen werden können.

Ohne das Back-annotation-Modul evaluieren die `if`-Bedingungen in Abb. 4.9 (`addJoinMapping` Zeile 14, `addRefMapping` Zeile 17 und `addDeadMapping` Zeile 7) zu `false`, sodass keine Back-annotation durchgeführt wird. So werden Abhängigkeiten zwischen den Modulen Back-annotation und ICD-LLIR vermieden.

4.2.2 Transformation von detaillierten WCET$_{est}$-Informationen

Nach der Herstellung der Verbindung zwischen IR- und LLIR-Basis-Blöcken kann nach der WCET-Analyse die Transformation von (detaillierten) WCET$_{est}$-Informationen von der ICD-LLIR in die ICD-C durchgeführt werden. Mit Hilfe des Wörterbuchs *LLIR- nach IR-Basis-Block* wird für jeden LLIR-Basis-Block der zugehörige IR-Basis-Block ermittelt. Anschließend werden die WCET$_{est}$-Informationen des LLIR-Basis-Blocks in das IR_PersistentObject des IR-Basis-Blocks übertragen, das über den Schlüssel `WCET` eindeutig identifiziert werden kann. Weil die Abbildung, die durch das Wörterbuch beschrieben wird, nicht injektiv ist, kann es sein, dass der IR-Basis-Block bereits WCET$_{est}$-Informationen enthält. In einem solchen Fall wird wie folgt vorgegangen:

- Die bereits angeheftete WCET$_{est}$ und die WCET$_{est}$ des betrachteten LLIR-Basis-Blocks werden aufsummiert.
- Die WCEC$_{sum}$ des betrachteten LLIR-Basis-Blocks wird übernommen, falls sie größer ist als die bereits vorhandene WCEC$_{sum}$.
- Falls das WCEP-Attribut des IR-Basis-Blocks den Wert `false` enthält, wird das WCEP-Attribut des LLIR-Basis-Blocks übernommen.
- Falls das Infeasible-Attribut des IR-Basis-Blocks den Wert `true` enthält, wird das Infeasible-Attribut des LLIR-Basis-Blocks übernommen.

Diese Vorgehensweise lässt sich am Beispiel der 1-zu-n-Beziehung begründen, die aufgrund von komplexen Bedingungen im Quellcode entsteht. Es wurde erwähnt, dass eine komplexe Bedingung mit mehreren Vergleichen durch einen IR-Basis-Block überdeckt wird. Zudem wurde betont, dass jeder Vergleich durch einen eigenen LLIR-Basis-Block überdeckt wird. Jeder dieser LLIR-Basis-Blöcke kann eine unterschiedliche WCET$_{est}$ aufweisen, weil die einzelnen Vergleiche unterschiedlich hohe Kosten verursachen können. Auch die Anzahl der Ausführungen der LLIR-Basis-Blöcke kann sich aufgrund der *Short Circuit Evaluation* unterscheiden, weil die Short Circuit Evaluation die Auswertung von booleschen Ausdrücken abbricht, wenn das Ergebnis bereits feststeht. Aus diesem Grund können sich auch die Attribute WCEP und Infeasible unterscheiden.

Anschließend kommt das Wörterbuch *IR-Basis-Block Referenzen* zum Einsatz. Für jeden IR-Basis-Block wird im zugehörigen IR_PersistentObject ein Verweis auf seinen Referenz IR-Basis-Block gespeichert, falls im Wörterbuch ein passender Eintrag existiert. Solche IR-Basis-Blöcke können über den Referenz IR-Basis-Block auf WCET$_{est}$-Informationen zugreifen.

Zum Schluss wird noch die Liste der IR-Basis-Blöcke abgearbeitet, deren zugehörige LLIR-Basis-Blöcke durch LLIR-Optimierungen gelöscht wurden. Auch solche IR-Basis-Blöcke erhalten ein IR_PersistentObject, das WCET$_{est}$-Informationen aufnehmen kann. Alle WCET$_{est}$-Informationen werden aber mit `0` oder `false` initialisiert.

Damit ist die Transformation von detaillierten WCET$_{est}$-Informationen abgeschlossen.

4.3 Integration des Back-annotation-Moduls in den WCC

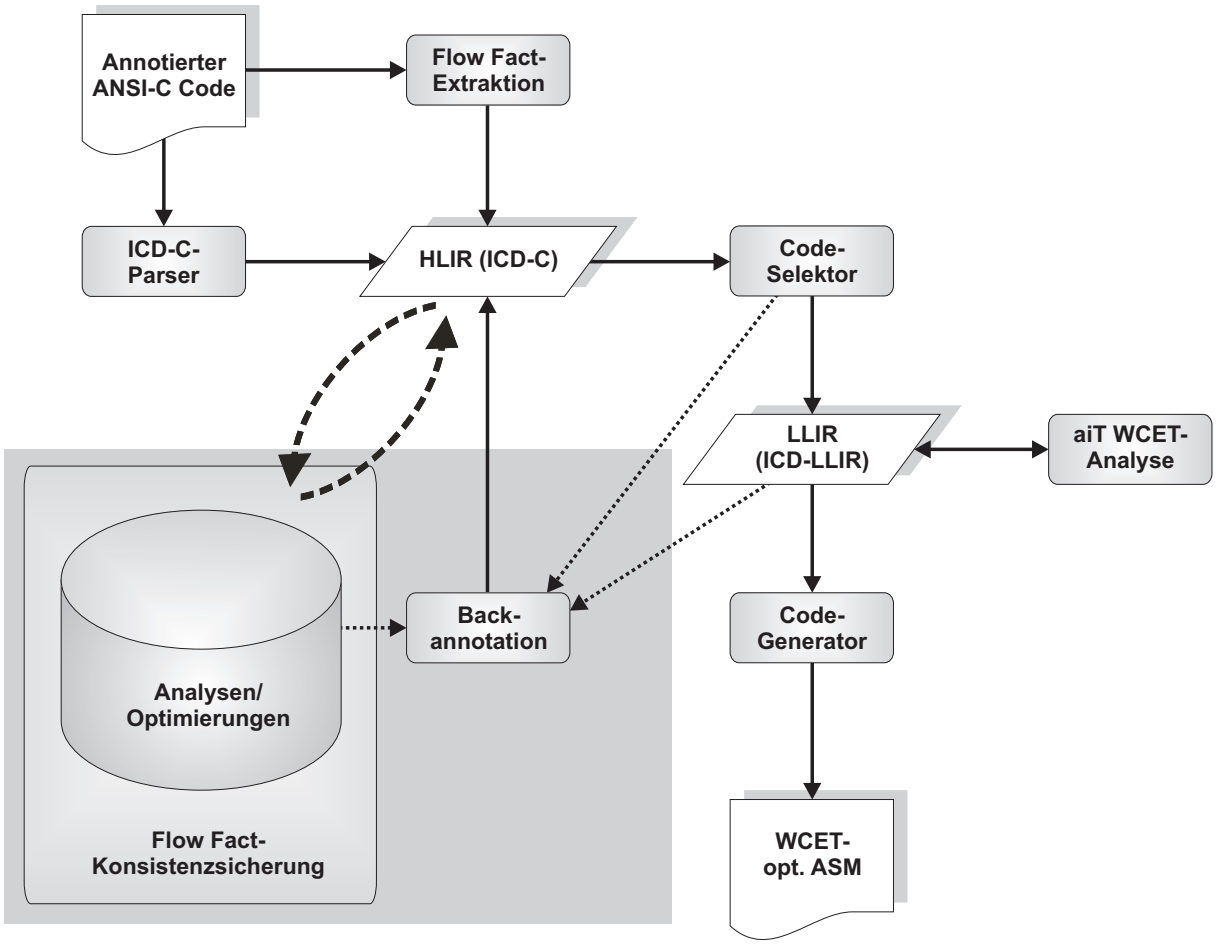


Abbildung 4.10: *Integration des Back-annotation-Moduls in den WCC*

Abb. 4.10 zeigt die Integration des Back-annotation-Moduls in den WCC. Aus Gründen der Übersichtlichkeit wird die Verbindung zwischen der LLIR (ICD-LLIR) und den Analysen/Optimierungen in dieser und in den folgenden Abbildungen des WCCs nicht dargestellt.

In Abb. 4.10 ist ersichtlich, dass das Back-annotation-Modul im Rahmen dieser Arbeit weiter ausgebaut wurde. Das Back-annotation-Modul wurde um eine Transformation von detaillierten $WCET_{est}$-Informationen erweitert. Die Verbindung zwischen IR- und LLIR-Basis-Blöcken wird im Code-Selektor hergestellt und muss im Laufe der LLIR-Optimierungen konsistent gehalten werden. Anschließend kann nach der WCET-Anlayse die Transformation von detaillierten $WCET_{est}$-Informationen von der ICD-LLIR in die ICD-C durchgeführt werden. Detaillierte $WCET_{est}$-Informationen werden in IR_PersistentObjects gespeichert und an jedem IR-Basis-Block angeheftet. Der Zugriff auf diese Informationen erfolgt über den Schlüssel `WCET`.

Das folgende Beispiel zeigt, wie innerhalb der ICD-C nach der Back-annotation-Phase auf die $WCET_{est}$ eines IR-Basis-Blocks zugegriffen werden kann:

```
1  IR_BasicBlock *bb = ...;
2
3  IR_WCETObject *wcetObj = dynamic_cast< IR_WCETObject * >
4                              ( bb->getUserData( "WCET" ) );
5
6  unsigned long long blockWCET = 0;
7
8  if ( wcetObj ) {
9    if ( wcetObj->getReferenceBlock() ) {
10     blockWCET = wcetObj->getReferenceWCET();
11   } else {
12     blockWCET = wcetObj->getWCET();
13   }
14 }
```

Abbildung 4.11: *Beispiel für das Auslesen der $WCET_{est}$ eines IR-Basis-Blocks*

In Abb. 4.11 wird in Zeile 4 mittels der Funktion `getUserData` über den Schlüssel `WCET` auf das IR_PersistentObject für $WCET_{est}$-Informationen zugegriffen. Zurückgegeben wird ein Objekt der Klasse IR_WCETObject, welches von der Klasse IR_PersistentObject erbt und für das Schreiben und Lesen von (detaillierten) $WCET_{est}$-Informationen zuständig ist. In Zeile 9 wird anschließend überprüft, ob der IR-Basis-Block, dessen $WCET_{est}$ ermittelt werden soll, einen anderen IR-Basis-Block referenziert (m-zu-1-Beziehung). In einem solchen Fall wird in Zeile 10 über `getReferenceWCET()` die $WCET_{est}$ des Referenz IR-Basis-Blocks ausgelesen. Ansonsten wird in Zeile 12 über `getWCET()` die eigene $WCET_{est}$ ausgelesen.

In sehr seltenen Fällen kann es vorkommen, dass aufgrund der Berechnung von aiT die $WCET_{est}$ eines (IR- bzw. LLIR-)Basis-Blocks gleich null ist, obwohl dieser nicht als `infeasible` gekennzeichnet wurde und in Wahrheit auf dem WCEP liegt. Seine $WCET_{est}$ wird anderswo (möglicherweise im Basis-Block des Vorgängers) berücksichtigt. Dieser Fall tritt potentiell nur bei den Basis-Blöcken auf, die eine sehr geringe $WCET_{est}$ aufweisen, und ist daher unkritisch.

4.4 Verifikation

Nach der Transformation von detaillierten WCET$_{est}$-Informationen wird die Verifikation durchgeführt. Die Verifikation findet nach jeder Back-annotation-Phase statt und besteht aus zwei Schritten:

Im ersten Schritt wird überprüft, ob alle IR-Basis-Blöcke ein IR_PersistentObject für WCET$_{est}$-Informationen besitzen. Falls ein IR-Basis-Block kein solches IR_PersistentObject besitzt, wird eine Warnung ausgegeben.

Im zweiten Schritt werden die WCET$_{est}$-Informationen aller IR-Basis-Blöcke BB aufsummiert und die Summe wird mit der globalen WCET$_{est}$ des zu analysierenden Programms P verglichen:

$$WCET_{est}(P) = \sum_{BB} WCET_{est}(BB) \tag{4.2}$$

Falls sich die Summen unterscheiden, wird ebenfalls eine Warnung ausgegeben.

Die Verifikation der transformierten WCET$_{est}$-Informationen konnte für alle Benchmarks in Kombination mit jeder Optimierungsstufe erfolgreich durchgeführt werden.

5 Cold Path

Dieses Kapitel präsentiert ein neues Verfahren mit dem Namen **Cold Path**. Der Cold Path demonstriert, wie detaillierte WCET_{est}-Informationen, die nach der Transformation aus Kapitel 4 auf High-Level IR-Ebene zur Verfügung stehen, weiter ausgewertet werden, damit High-Level Compiler-Optimierungen von noch mehr Informationen profitieren können. Detaillierte WCET_{est}-Informationen werden in diesem Kapitel erstmals eingesetzt.

5.1 Motivation

Optimierungen, die eine Reduktion der ACET erreichen wollen und daher auf dem häufig ausgeführten Pfad (ACEP) eingesetzt werden, haben einen großen Vorteil gegenüber WCET-Optimierungen: Der häufig ausgeführte Pfad eines Programms ändert sich nie. Der längste Ausführungspfad eines Programms (WCEP) hingegen kann im Laufe der Transformationen durch Optimierungen wechseln. Dieser Umstand wird in Abb. 5.1 auf Seite 42 veranschaulicht.

In Abb. 5.1 – (1) ist eine Funktion mit dem Namen `foo` dargestellt, die eine `if-else`-Verzweigung enthält. Die `if-else`-Verzweigung wird durch den einzigen Parameter der Funktion gesteuert. Der abstrahierte Kontrollflussgraph der Funktion ist in Abb. 5.1 – (2) skizziert. Die gestrichelten Pfeile symbolisieren die möglichen Ausführungspfade.

Anschließend wird auf diesem Code die aiT WCET-Analyse durchgeführt. Ein mögliches Ergebnis der WCET-Analyse zeigt Abb. 5.1 – (3). Die WCET_{est} des `then`-Blocks der `if-else`-Verzweigung beträgt 90 Zyklen und die des `else`-Blocks beträgt 70 Zyklen. Die WCET_{est} der anderen Basis-Blöcke ist an dieser Stelle nicht relevant. Das Programm aiT hat zudem ermittelt, dass sich der `then`-Block auf dem WCEP befindet und der `else`-Block nicht. Im Folgenden symbolisieren durchgezogene Pfeile die gerichteten Kanten in einem Kontrollflussgraph, die auf dem WCEP liegen. Gepunktete Pfeile hingegen markieren die gerichteten Kanten in einem Kontrollflussgraph, die *nicht* auf dem WCEP liegen.

Ein Vorhaben dieser Diplomarbeit ist die Entwicklung von Optimierungen, die eine WCET-Minimierung zum Ziel haben. Um die WCET eines Programms zu minimieren, muss sich die Optimierung auf die Teile des Quellcodes konzentrieren, die auf dem WCEP liegen. Eine andere Strategie führt in der Regel nicht zu einer Verbesserung der WCET. Aus diesem Grund würde sich eine WCET-gesteuerte Optimierung um eine Verbesserung der WCET_{est} des `then`-Blocks in Abb. 5.1 – (3) bemühen.

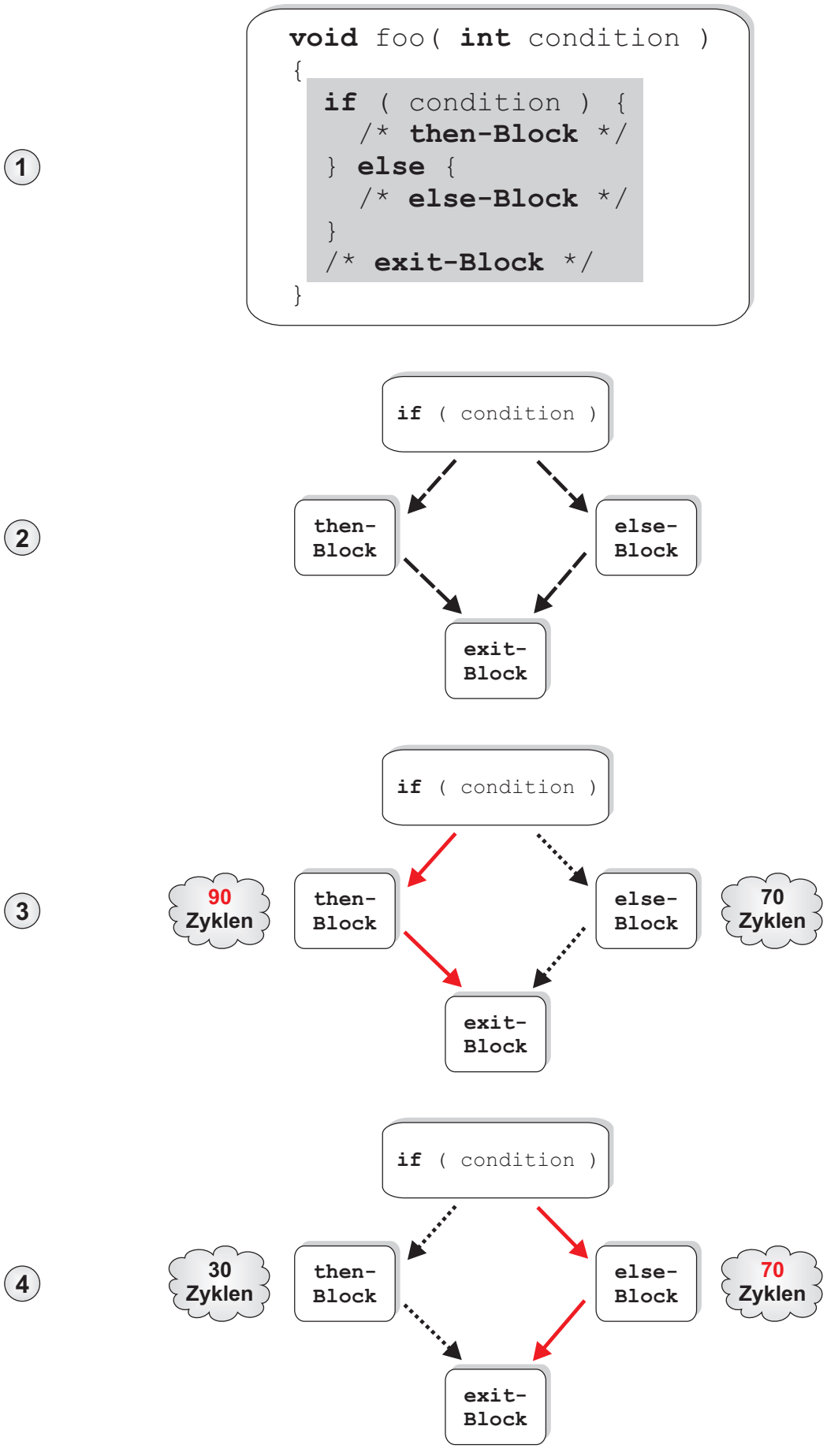


Abbildung 5.1: *Beispiel für sich verändernde WCEPs*

In Abb. 5.1 – (4) ist die Situation nach der Optimierung zu erkennen. Die neue $WCET_{est}$ des optimierten `then`-Blocks beträgt nach der Optimierung nur noch 30 Zyklen. Demnach wurde die $WCET_{est}$ des `then`-Blocks um 60 Zyklen reduziert. Insgesamt hat sich die $WCET_{est}$ der `if-else`-Verzweigung aber nur um 20 Zyklen (90 Zyklen - 70 Zyklen) verbessert, weil der WCEP aufgrund der Optimierung gewechselt hat. Der `else`-Block liegt nun auf dem WCEP, weil er eine größere $WCET_{est}$ aufweist. Das Beispiel zeigt, dass der WCEP im Laufe der Transformationen durch Optimierungen wechseln kann.

5.1.1 Optimierungsstrategien zur WCET-Minimierung

Optimierungsstrategien zur WCET-Minimierung müssen damit rechnen, dass sich der Pfad nach jedem Optimierungsschritt ändern kann. Daher müssen Optimierungen die Auswirkungen ihrer Entscheidungen nicht nur auf lokaler, sondern auch auf globaler Ebene berücksichtigen.

Eine einfache Optimierungsstrategie zur WCET-Minimierung, die typischerweise in vielen Verfahren zum Einsatz kommt, besteht darin, eine neue WCET-Analyse nach jedem Optimierungsschritt durchzuführen, sodass im nächsten Schritt auf dem gültigen WCEP optimiert werden kann. Diese einfache Strategie wird z.B. in [Lokuciejewski u.a. 2008a], [Puaut 2006] und [Zhao u.a. 2004] verfolgt. Eine WCET-Analyse ist aber sehr zeitaufwändig und erhöht drastisch die Kompilierzeit, weswegen sich WCET-Optimierungen in der Praxis als untauglich erweisen können. Daher sollte eine Pfadaktualisierung nach Möglichkeit vermieden werden. Dies stellt die Hauptmotivation für den Cold Path dar. Nachfolgend wird der Cold Path definiert.

Cold Path

Definition 1 *Der Cold Path liegt auf dem WCEP und beschreibt die Teilpfade, die immer auf dem WCEP bleiben.*

Der Name deutet darauf hin, dass der Cold Path im Gegensatz zum WCEP nicht wechseln kann. Dieses stabile Verhalten wird durch Kälte symbolisiert. Außerdem stellt der Name ein Wortspiel mit dem Begriff *Hot Path* dar, welcher bekanntlich die oft benutzten Code-Teile bezeichnet.

Cold Path-Informationen ermöglichen die Entwicklung von besseren Optimierungsstrategien zur WCET-Minimierung. Optimierungen, die auf dem Cold Path ausgeführt werden, müssen anschließend keine neue WCET-Analyse starten, um zu ermitteln, ob der WCEP gewechselt hat, weil der Cold Path im Gegensatz zum WCEP ein stabiles Verhalten aufweist. Das vorgestellte Verfahren ist zudem unabhängig von einer bestimmten Programmiersprache oder Zwischendarstellung. Dennoch wird im Folgenden die höhere Programmiersprache C benutzt, um konkrete Programmbeispiele angeben zu können. Außerdem wird in dieser Arbeit der Cold Path-Algorithmus nur für die High-Level IR ICD-C implementiert, weil diese Zwischendarstellung für die vorliegende Arbeit von zentraler Bedeutung ist.

In Abschnitt 5.2 wird der Cold Path-Algorithmus vorgestellt. In Abschnitt 5.5 wird der Cold Path von Realworld-Benchmarks berechnet und die Ergebnisse werden diskutiert. Abschnitt 5.4 beschreibt schließlich die Verifikation der Cold Path-Information.

5.2 Algorithmus

Im letzten Abschnitt wurde betont, dass eine Verbesserung auf dem WCEP dazu führen kann, dass ein anderer Pfad der neue WCEP wird. Daher ist das Ziel des Cold Path-Algorithmus das Finden von Teilpfaden auf dem WCEP, die auch nach Transformationen durch Optimierungen immer auf dem WCEP bleiben. Solche Teilpfade liegen nach Definition 1 auf dem Cold Path und beschreiben Regionen auf dem WCEP, in denen es das Problem der sich verändernden WCEPs nicht gibt. Im Folgenden wird die Cold Path-Berechnung nur auf IR-Basis-Block-Ebene demonstriert, weil detaillierte WCET_{est}-Informationen derzeit nur auf dieser Ebene verfügbar sind. WCET_{est}-Informationen von einzelnen Anweisungen oder Ausdrücken sind nicht bekannt. Welche Nachteile diese Vorgehensweise mit sich bringt, wird weiter unten erläutert.

Das Problem der sich verändernden WCEPs wird verursacht durch Verzweigungen im Programm. Gäbe es keine Verzweigungen wie z.B. `if`-, `if-else`- oder `switch`-Verzweigungen, gäbe es auch das Problem nicht. Daraus lässt sich folgende Regel ableiten:

Regel 1 *IR-Basis-Blöcke, die auf dem WCEP liegen und nicht Teil einer Verzweigung sind, liegen auf dem Cold Path.*

Abb. 5.2 – (1) zeigt den Kontrollflussgraph eines Programms, das eine Verzweigung enthält. Regel 1 besagt, dass die IR-Basis-Blöcke vor und nach der Verzweigung auf dem Cold Path liegen (IR-Basis-Blöcke 1, 2, 5 und 6). Im Folgenden symbolisieren Pfeile mit zwei Pfeilspitzen Teilpfade, die auf dem Cold Path liegen.

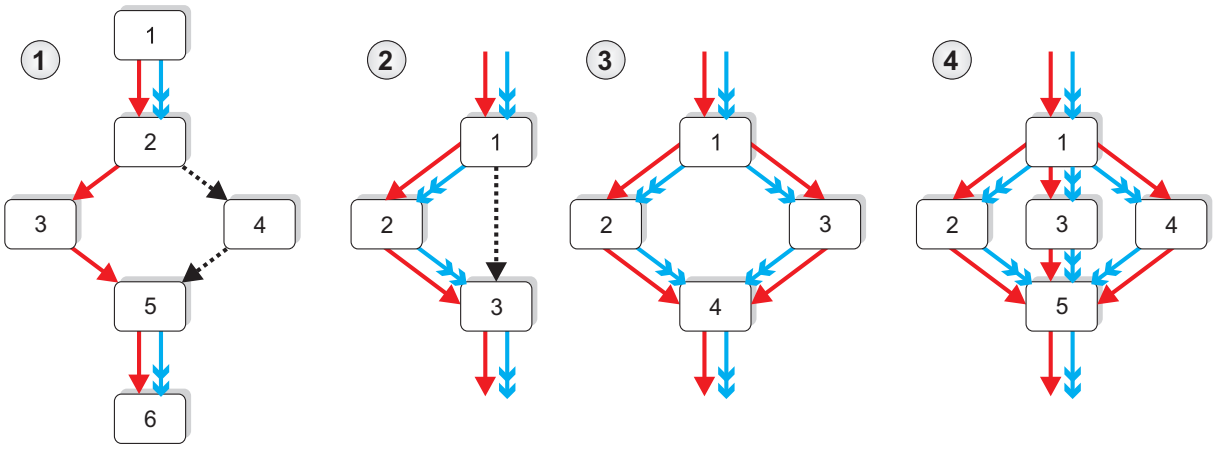


Abbildung 5.2: *Cold Path-Berechnungsbeispiele*

Die Verzweigung in Abb. 5.2 – (1) ist ein Beispiel für eine Verzweigung, die anfällig für einen

Pfadwechsel ist. Dabei wird angenommen, dass IR-Basis-Block 4 nicht unausführbar ist. Nur ein Ausführungspfad der Verzweigung liegt auf dem WCEP (IR-Basis-Block 3). Eine solche Situation kann entstehen, wenn die WCET-Analyse die pessimistische Annahme trifft, dass in jedem Kontext der zugehörigen Funktion der längere Ausführungspfad der Verzweigung berücksichtigt wird. Kontexte wurden erstmals in Kapitel 2 eingeführt und spielen bei der Cold Path-Berechnung eine wichtige Rolle.

Verzweigungen, die anfällig für einen Pfadwechsel sind, werden im Folgenden **bösartige Verzweigungen** genannt. Das wohl wichtigste Ergebnis in diesem Kapitel ist die Erkenntnis, das nicht jede Verzweigung anfällig für einen Pfadwechsel ist. Solche Verzweigungen werden im Folgenden **gutartige Verzweigungen** genannt. Regel 2 charakterisiert gutartige Verzweigungen:

Regel 2 *Falls der erste IR-Basis-Block jedes möglichen Ausführungspfades einer Verzweigung entweder auf dem WCEP liegt oder unausführbar ist, so ist die Verzweigung gutartig. Andernfalls ist die Verzweigung bösartig.*

In Abb. 5.2 – (2), (3) und (4) sind Beispiele für gutartige Verzweigungen skizziert.

Die Verzweigung in Abb. 5.2 – (2) repräsentiert z.B. eine `if`-Verzweigung. In IR-Basis-Block 1 wird die `if`-Bedingung ausgewertet, IR-Basis-Block 2 entspricht dem `then`-Block und IR-Basis-Block 3 entspricht dem `exit`-Block. Dabei handelt es sich um eine gutartige Verzweigung, weil sowohl der `then`-Block als auch der `exit`-Block auf dem WCEP liegen. Zu einem Pfadwechsel kann es an dieser Stelle nicht kommen, wenn angenommen werden kann, dass die WCET_{est} des `then`-Blocks immer größer ist, als die WCET_{est}, die entsteht, wenn der `then`-Block nicht genommen wird.

Die Verzweigung in Abb. 5.2 – (3) beschreibt z.B. eine `if-else`-Verzweigung. In diesem Fall liegt sowohl der `then`-Block als auch der `else`-Block auf dem WCEP und damit auch auf dem Cold Path. Situationen wie diese können auftreten, wenn die WCET-Analyse unterschiedliche Kontexte von Funktionen und Schleifen berücksichtigt. Auch in einer solchen Situation kann der Pfad nicht wechseln. Um zu verstehen, warum es nicht zu einem Pfadwechsel kommen kann, wird anhand des folgenden Beispiels das Konzept der Kontexte näher vorgestellt.

```
1  int foo( int condition )
2  {
3    if ( condition ) {
4      /* then-Block */
5    } else {
6      /* else-Block */
7    }
8    /* exit-Block */
9  }
10
11 int main()
12 {
13   return foo( 1 ) + foo( 0 );
14 }
```

Abbildung 5.3: *Betrachtung unterschiedlicher Funktionskontexte*

In Kapitel 2 wurde beschrieben, dass Kontexte die Genauigkeit der WCET-Abschätzung erhöhen können. Zudem wurde betont, dass Funktionskontexte eine getrennte Betrachtung der Funktionsaufrufe einer bestimmten Funktion ermöglichen. Im Beispielprogramm aus Abb. 5.3 wird zum Einen verdeutlicht, wie es zu der Situation, die in Abb. 5.2 – (3) dargestellt ist, kommen kann, zum Anderen wird demonstriert, wie die Betrachtung unterschiedlicher Kontexte einer Funktion einen Pfadwechsel unterbinden kann.

Das Beispielprogramm besteht aus zwei Funktionen: `main` und `foo`. Die Funktion `foo` enthält eine `if-else`-Verzweigung, welche durch den einzigen Parameter der Funktion gesteuert wird. Die Funktionsaufrufe `foo( 1 )` und `foo( 0 )` in Zeile 13 führen dazu, dass bei jedem Programmlauf beide Ausführungspfade der `if-else`-Verzweigung ausgeführt werden.

Eine einfache WCET-Analyse, die keine Kontexte unterstützt, geht davon aus, dass der längere Ausführungspfad der `if-else`-Verzweigung auf dem WCEP liegt. Diese Annahme ist zwar sicher, führt aber zu einer WCET-Überabschätzung, weil die WCET-Analyse bei jedem Aufruf der Funktion `foo` den längeren Ausführungspfad der `if-else`-Verzweigung berücksichtigt. Eine intelligentere WCET-Analyse kann mit Hilfe von Kontexten die beiden Funktionsaufrufe zu der Funktion `foo` unterscheiden. Wenn die WCET-Analyse für jeden Funktionskontext die Bedingung der `if-else`-Verzweigung statisch auswerten kann, dann kann sie anschließend schlussfolgern, dass beide Ausführungspfade der `if-else`-Verzweigung auf dem WCEP liegen. Dies führt zu einer Verbesserung der WCET-Abschätzung. Je mehr Kontexte bei der WCET-Analyse unterschieden werden, desto genauer ist potentiell das Ergebnis der WCET-Abschätzung.

Außerdem wird ersichtlich, dass der WCEP in einer solchen Situation nicht wechseln kann, weil eine WCET-Analyse, die Kontexte unterstützt, jedesmal beide Ausführunspfade berücksichtigt, um eine möglichst genaue WCET-Abschätzung durchführen zu können. Dieser Nebeneffekt wird durch den Cold Path-Algorithmus ausgenutzt. Somit gilt: Je mehr Kontexte bei der WCET-Analyse unterschieden werden, desto genauer ist potentiell das Ergebnis der WCET-Abschätzung *und* desto geringer ist potentiell die Gefahr eines Pfadwechsels.

Die gutartige Verzweigung in Abb. 5.2 – (4) entspricht z.B. einer `switch`-Anweisung. Eine `switch`-Anweisung ist eine verallgemeinerte Form der `if-else`-Anweisung und wird analog zu den obigen Beispielen behandelt. Diese Vorgehensweise wird in Regel 2 verallgemeinert.

Mit Hilfe dieser Regeln kann der Cold Path rekursiv berechnet werden. Der Cold Path-Algorithmus besteht aus drei Schritten und beginnt mit der Funktion `main`.

- IR-Basis-Blöcke, die auf dem WCEP liegen und nicht Teil einer Verzweigung sind, liegen auf dem Cold Path.
- Bei gutartigen Verzweigungen wird rekursiv der Cold Path aller ausführbaren Zweige berechnet. Somit werden alle IR-Basis-Blöcke betrachtet, die Teil einer solchen Verzweigung sind.
- Funktionsaufrufe, die sich auf dem Cold Path befinden und *zwangsläufig* ausgeführt werden, werden gesammelt und der Cold Path der zugehörigen Funktion wird berechnet, falls die auf-

gerufene Funktion noch nicht betrachtet wurde.

Ein Funktionsaufruf wird zwangsläufig ausgeführt, wenn der Funktionsaufruf in seinem zugehörigen Ausdrucks-Baum garantiert ausgeführt wird. Beispiel:

```
int erg = foobar() ? foo() : bar();
```

Abbildung 5.4: *Beispiel für einen zwangsläufig ausgeführten Funktionsaufruf*

Der Funktionsaufruf `foobar()` wird in seinem Ausdrucks-Baum garantiert ausgeführt, wohingegen die Funktionsaufrufe `foo()` und `bar()` nicht zwangsläufig ausgeführt werden. Abhängig vom Ergebnis von `foobar()` wird entweder `foo()` oder `bar()` ausgeführt. Das Wissen darüber, ob ein Funktionsaufruf garantiert ausgeführt wird oder nicht, ist notwendig, weil der WCEP und damit auch der Cold Path im WCC derzeit nur auf Basis-Block-Ebene betrachtet werden können. Die obige Anweisung wird durch einen einzigen IR-Basis-Block b überdeckt. Falls b auf dem Cold Path liegt, dann liegen auch alle Anweisungen, die in b enthalten sind (insbesondere auch die Anweisung in Abb. 5.4), auch auf dem Cold Path. Damit liegt auch die Funktion `foobar` auf dem Cold Path, weil es sich bei `foobar()` um einen Funktionsaufruf handelt, der in b zwangsläufig ausgeführt wird. Aus der Anweisung in Abb. 5.4 lässt sich aber nicht schlussfolgern, dass die Funktionen `foo` und `bar` auf dem Cold Path liegen, weil die zugehörigen Funktionsaufrufe in der obigen Anweisung nicht zwangsläufig ausgeführt werden. Dadurch wird eine Cold Path-Berechnung für diese beiden Funktionen unterbunden. Funktionsaufrufe in Zusammenhang mit dem Bedingungs-Operator treten aber in Realworld-Benchmarks sehr selten auf, sodass diese Beschränkung vernachlässigbar ist.

Auch bei der Auswertung von booleschen Ausdrücken spielt es eine Rolle, ob ein Funktionsaufruf zwangsläufig ausgeführt wird oder nicht, weil die Short Circuit Evaluation die Auswertung von booleschen Ausdrücken abbricht, wenn das Ergebnis bereits feststeht. Außerdem wird wieder die gesamte Bedingung durch einen einzigen IR-Basis-Block überdeckt (siehe Kapitel 4). Abb. 5.5 zeigt ein Beispiel:

```
if ( ( a < b ) && foo() && bar() ) {
  /* then-Block */
}
```

Abbildung 5.5: *Beispiel für Funktionsaufrufe in Bedingungen*

Falls der erste Ausdruck der `if`-Bedingung zu `false` evaluiert, müssen die Funktionsaufrufe `foo()` und `bar()` nicht mehr ausgeführt werden. Daher betrachtet der Algorithmus in Schritt 3 nur die Funktionsaufrufe, die zwangsläufig ausgeführt werden.

Es hat sich jedoch gezeigt, dass Funktionsaufrufe, die nicht an erster Stelle einer Bedingung stehen, in Realworld-Benchmarks sehr selten auftreten.

5.2.1 Transformation der Cold Path-Informationen in die LLIR

Weil sich diese Arbeit nicht mit der Entwicklung von Low-Level Compiler-Optimierungen beschäftigt, wurde der Cold Path nur für die High-Level IR ICD-C implementiert. Cold Path-Informationen stehen daher nur in der ICD-C auf Basis-Block-Ebene zur Verfügung. Ein Algorithmus für die Low-Level IR ICD-LLIR wurde nicht entwickelt. Stattdessen wurde versucht, die Cold Path-Informationen von der High-Level IR in die Low-Level IR zu transformieren. Eine solche Transformation ist aber aufgrund der m-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken problematisch (siehe Kapitel 4). Daher wurde in dieser Arbeit eine einfache Transformation entwickelt, die zumindest für die 1-zu-1- und m-zu-1-Beziehung zwischen IR- und LLIR-Basis-Blöcken einen Informationsaustausch gewährleistet. Die Transformation konnte jedoch aufgrund von technischen Restriktionen nicht verifiziert werden, weswegen im Folgenden nicht weiter darauf eingegangen wird.

5.3 Integration der Cold Path-Analyse in den WCC

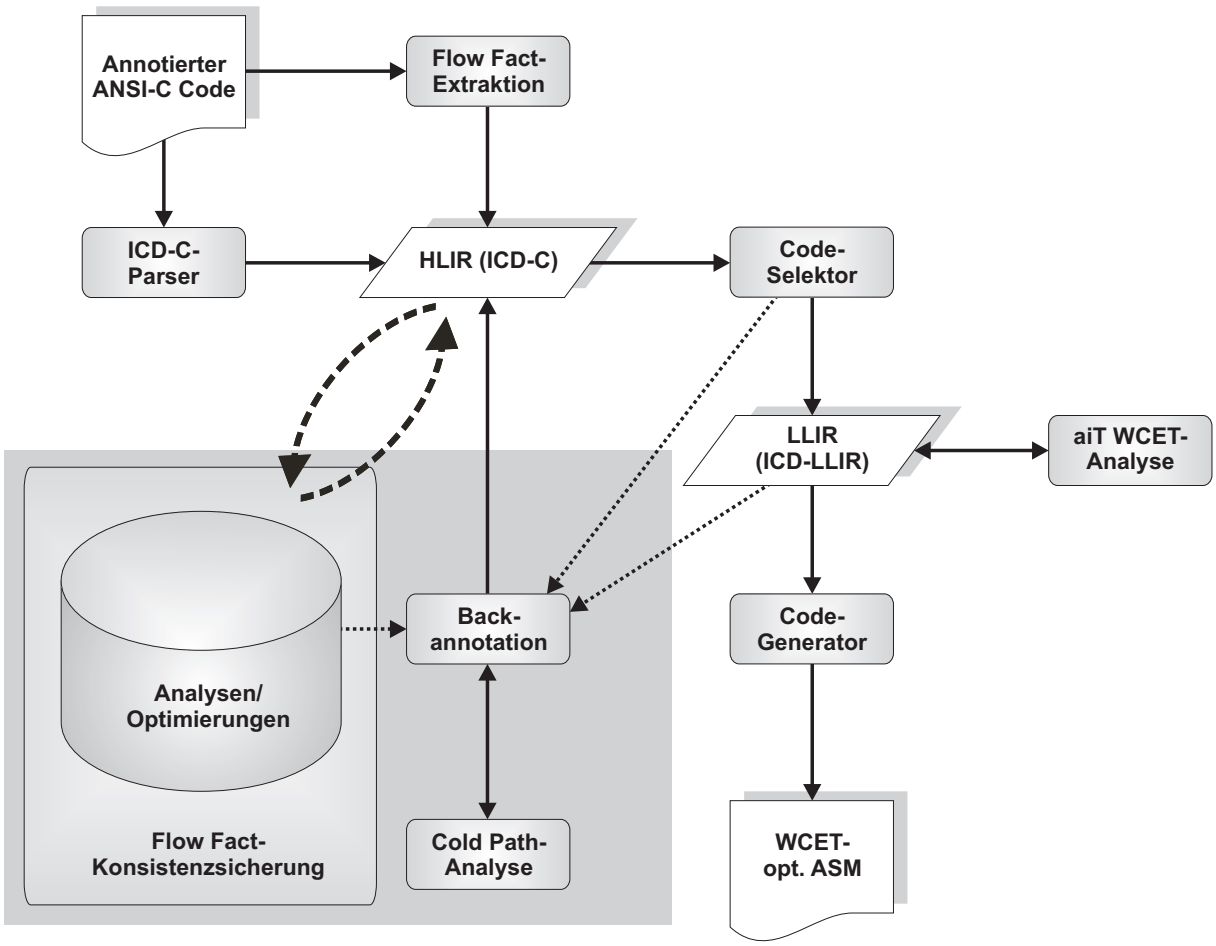


Abbildung 5.6: *Integration der Cold Path-Analyse in den WCC*

Abb. 5.6 veranschaulicht die Integration der Cold Path-Analyse in den WCC. Die Cold Path-Analyse macht Gebrauch von detaillierten WCET_{est}-Informationen, die nach der Transformation aus Kapitel 4

auf High-Level IR-Ebene zur Verfügung stehen. Daher wird die Cold Path-Analyse direkt im Anschluss der Transformation aufgerufen und wird im Folgenden als Teil der Back-annotation-Phase angesehen.

Die Cold Path-Information eines jeden IR-Basis-Blocks wird in demselben IR_PersistentObject gespeichert, wo auch die detaillierten WCET_{est}-Informationen des IR-Basis-Blocks gespeichert sind. Somit kann über den Schlüssel `WCET` sowohl auf WCET_{est}- als auch auf Cold Path-Informationen zugegriffen werden.

5.4 Verifikation

Bei der Verifikation der Cold Path-Informationen wurden die nach Definition 1 erforderlichen Cold Path-Eigenschaften überprüft.

Zum Einen wurde überprüft, ob der Cold Path auf dem WCEP liegt. Liegt ein IR-Basis-Block auf dem Cold Path, so muss er auch auf dem WCEP liegen. Diese Eigenschaft wird durch den ersten Schritt des Cold Path-Algorithmus sichergestellt.

Zum Anderen wurde kontrolliert, ob der Cold Path anfällig für einen Pfadwechsel ist. Für diesen Zweck wurden die Verzweigungen in einem Programm näher untersucht, weil das Problem der sich verändernden WCEPs durch Verzweigungen im Programm verursacht wird. Das Programm wurde zuerst mit der Optimierungsstufe Oi ($i \in \{0, 1, 2\}$) übersetzt. In der Back-annotation-Phase wurde für jede Verzweigung festgehalten, ob sie gut- oder bösartig ist. Anschließend wurde das Programm mit einer höheren Optimierungsstufe Oj ($j \in \{1, 2, 3\} \wedge j > i$) erneut übersetzt. Danach fand die eigentliche Verifikation statt: Aus einer ursprünglich gutartigen Verzweigung darf nach der zweiten Optimierungsphase keine bösartige Verzweigung werden. Eine Verletzung dieser Aussage ist gleichbedeutend mit einem Pfadwechsel. Aus einer ursprünglich bösartigen Verzweigung jedoch kann nach der zweiten Optimierungsphase eine gutartige Verzweigung werden, weil das Programm in der zweiten Optimierungsphase mit einer höheren Optimierungsstufe übersetzt wird. So können z.B. bestimmte Ausführungspfade einer Verzweigung entfernt oder durch aiT als unausführbar gekennzeichnet werden.

Die Verifikation konnte für alle betrachteten Benchmarks erfolgreich durchgeführt werden.

5.5 Statistiken

Im Rahmen dieser Diplomarbeit wurde der Cold Path-Algorithmus auf High-Level IR-Ebene für den WCC implementiert und verifiziert. Anschließend wurde der Cold Path von Realworld-Benchmarks berechnet. Die Benchmarks stammen aus den speziell für Eingebettete Systeme entwickelten Benchmark-Suiten MRTC [MRTC 2008], MediaBench [Lee u.a. 1997] und NetBench [Memik u.a. 2001].

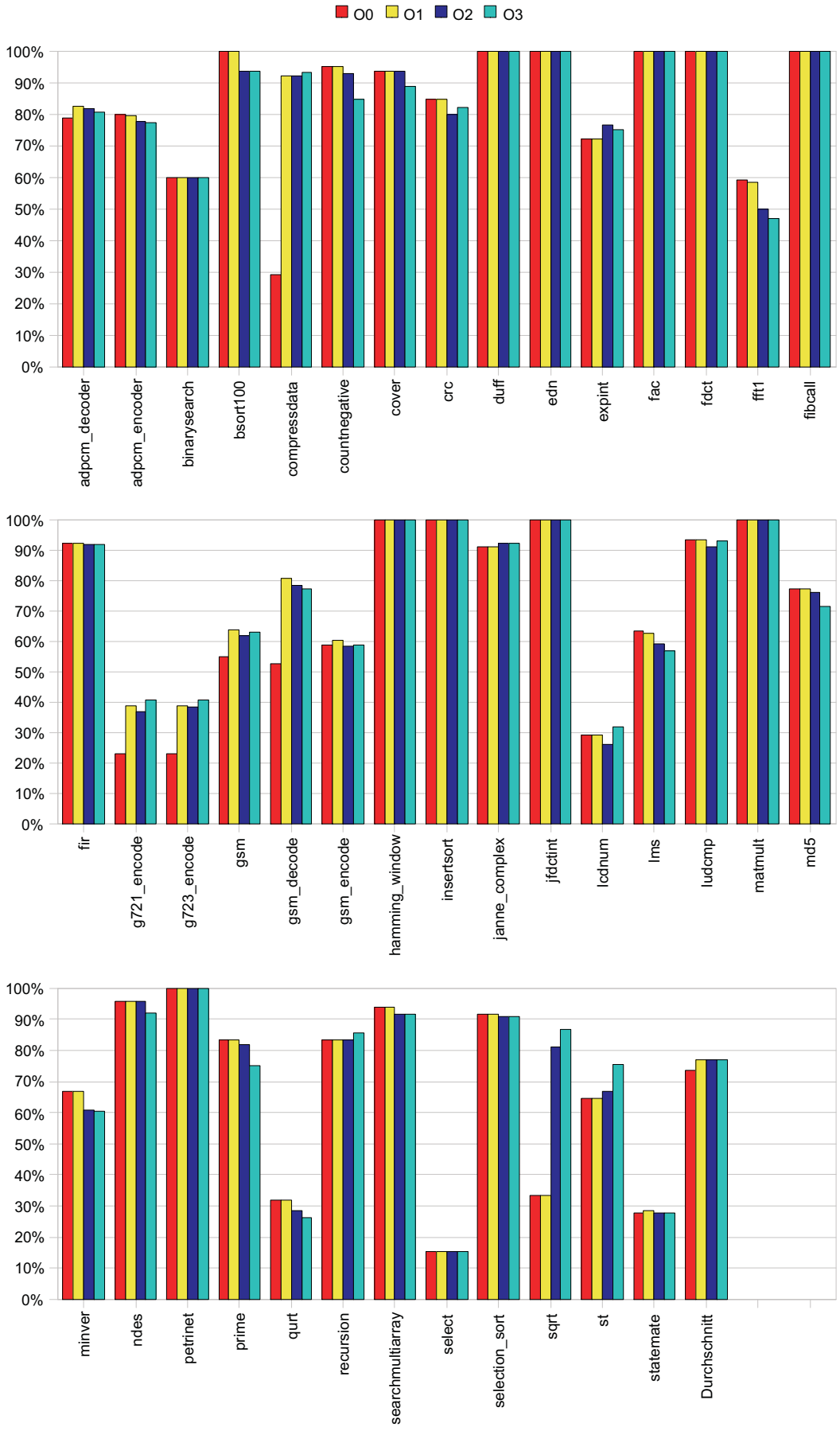


Abbildung 5.7: *Anteil der IR-Basis-Blöcke auf dem Cold Path*

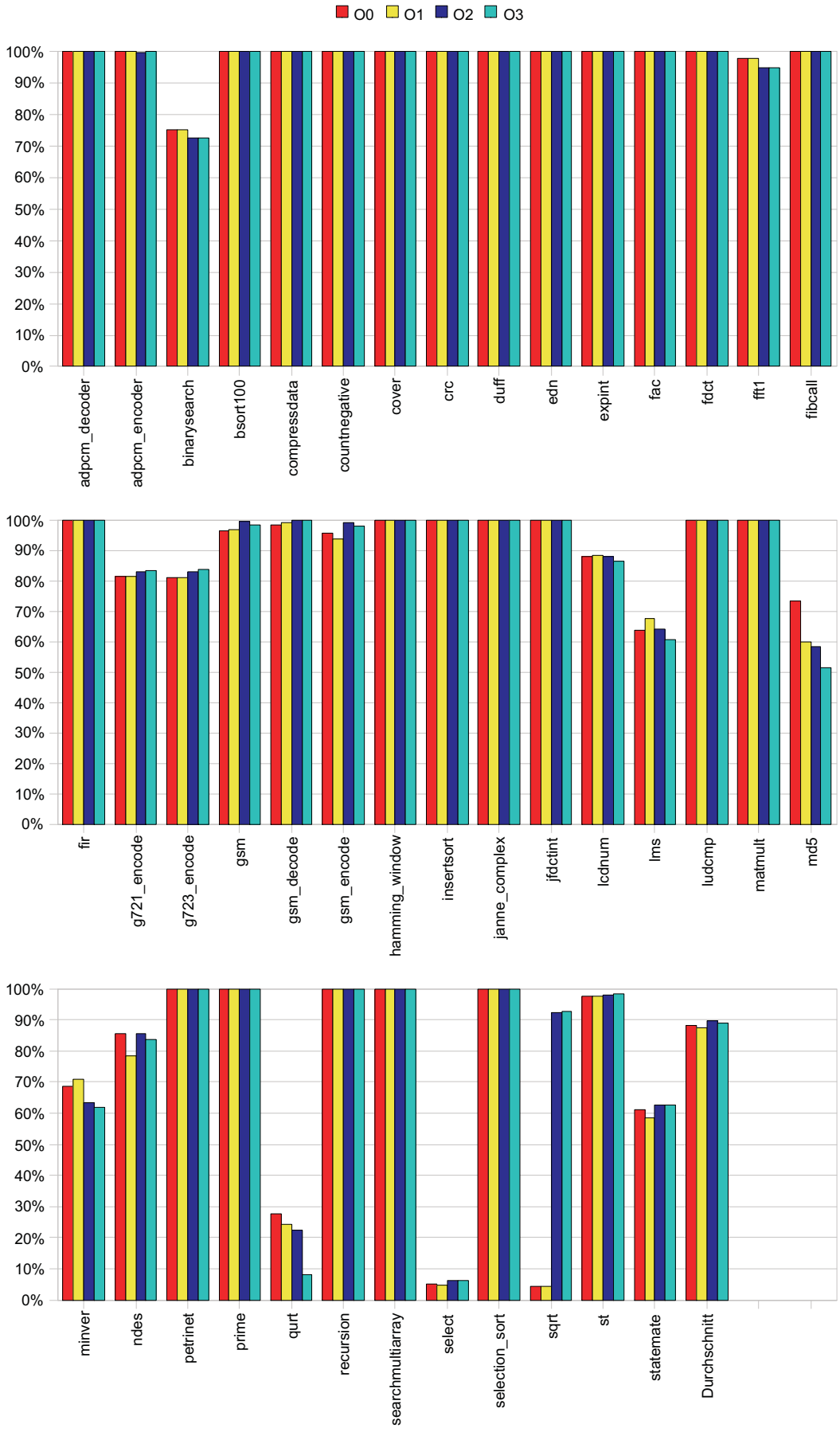


Abbildung 5.8: *Anteil der Zyklen auf dem Cold Path*

Für jeden einzelnen Benchmark und für jede Optimierungsstufe wurde der statische und dynamische Anteil an Quellcode auf dem Cold Path berechnet. Der statische Anteil entspricht dem Anteil der IR-Basis-Blöcke auf dem Cold Path und ist in Abb. 5.7 zu sehen.

Der dynamischen Anteil hingegen ergibt sich aus der Summe der WCET_{est} aller IR-Basis-Blöcke BB eines Benchmarks, die auf dem Cold Path liegen, dividiert durch die globale WCET_{est} des Benchmarks:

$$\frac{\sum_{BB} WCET_{est}(BB) * CP(BB)}{\sum_{BB} WCET_{est}(BB)}, \text{ wobei } CP(BB) = \begin{cases} 1, & BB \text{ liegt auf dem Cold Path} \\ 0, & \text{sonst} \end{cases} \tag{5.1}$$

Somit stellt der dynamische Anteil den Anteil der Zyklen auf dem Cold Path dar und ist in Abb. 5.8 zu sehen.

Auf dem ersten Blick fällt auf, dass ein Großteil der IR-Basis-Blöcke auf dem Cold Path liegen. Bei einigen Benchmarks wie z.B. edn, fac, fdct, fibcall, insertsort und jfdctint liegen sogar alle IR-Basis-Blöcke auf dem Cold Path. Folglich existiert das Problem der sich verändernde WCEPs nicht für diese Benchmarks. An dieser Stelle muss aber erwähnt werden, dass es sich bei vielen Benchmarks um kleine Benchmarks handelt, die zum Teil sogar keine Verzweigungen enthalten.

Beim Benchmark select hingegen liegen relativ wenige IR-Basis-Blöcke auf dem Cold Path, weil die meiste Zeit der Programmausführung in einer bösartigen `if-else`-Verzweigung stattfindet.

Bei einigen Benchmarks wie z.B. lms, prime und qurt führt eine höhere Optimierungsstufe zu einem Anstieg der Anzahl der IR-Basis-Blöcke, die *nicht* auf dem Cold Path liegen, wohingegen bei anderen Benchmarks wie z.B. g721_encode, g723_encode und sqrt genau das Gegenteil beobachtbar ist. Im Folgenden werden mögliche Ursachen genannt.

Code-größen-kritische Optimierungen wie z.B. Loop Unswitching, Loop Unrolling, Function Inlining und Procedure Cloning können zu einer Erhöhung der Anzahl der IR-Basis-Blöcke führen, die *nicht* auf dem Cold Path liegen. Falls die code-größen-kritische Optimierung auf dem Teil des WCEPs ausgeführt wird, der anfällig für einen Pfadwechsel ist und somit nicht auf dem Cold Path liegt, erhöht sich aufgrund der Optimierung die Anzahl der IR-Basis-Blöcke, die nicht auf dem Cold Path liegen. So wird z.B. beim Benchmark lms aufgrund der Optimierung Function Inlining der Funktionsrumpf einer Funktion, die auf dem Cold Path liegt, in eine Region kopiert, die nicht auf dem Cold Path liegt. Damit liegen automatisch alle IR-Basis-Blöcke des kopierten Funktionsrumpfes nicht auf dem Cold Path.

Andererseits können Optimierungen den Quellcode vereinfachen. In Abschnitt 3.1.1.4 wurde z.B. erwähnt, dass die Optimierung Fold Constant Code den Quellcode so weit vereinfachen kann, dass Verzweigungen mit konstant falscher Bedingung entfernt werden können. Falls es sich dabei um eine bösartige Verzweigung handelt, erhöht sich natürlich der Anteil der IR-Basis-Blöcke, die auf dem Cold Path liegen. Eine weitere Optimierung, die den Quellcode vereinfacht und auf höherer Optimierungsstufe (O2) aufgerufen wird, ist Eliminate Return Value. Beim Benchmark sqrt wird aufgrund dieser

Optimierung der `then`-Block einer bösartigen `if-else`-Verzweigung entfernt, sodass ab der Optimierungsstufe O2 aus einer bösartigen `if-else`-Verzweigung eine gutartige `if`-Verzweigung wird (vgl. Abb. 5.7 und 5.8).

Beim Vergleich der Abb. 5.7 und 5.8 fällt ebenfalls auf, dass z.B. beim Benchmark countnegative 100% der Zyklen auf dem Cold Path ausgeführt werden, obwohl nicht alle IR-Basis-Blöcke auf dem Cold Path liegen. IR-Basis-Blöcke, die als unausführbar gekennzeichnet wurden oder unerreichbaren Code darstellen, liegen natürlich nicht auf dem Cold Path, weil sie nicht auf dem WCEP liegen. Damit haben diese auch keinen Einfluss auf die globale WCET_{est} eines Benchmark. Aus diesem Grund ist der statische Anteil an Quellcode auf dem Cold Path im Durchschnitt geringer als der dynamische Anteil. Eine Ausnahme stellt der Benchmark select dar. Dieser Benchmark beinhaltet viele bösartige Verzweigungen, die zudem in Schleifen eingebettet sind.

Zusammengefasst lässt sich sagen, dass die Anzahl der WCET-Analysen stark reduziert werden kann, weil im Durchschnitt mehr als $2/3$ des Quellcodes auf dem Cold Path liegt.

6 WCET-gesteuertes Function Inlining

In diesem und im nächsten Kapitel geht es um die Ausnutzung von detaillierten WCET_{est}-Informationen und Cold Path-Informationen für traditionelle High-Level-Optimierungen, die üblicherweise eine Reduktion der ACET zum Ziel haben (vgl. Abb. 6.1).

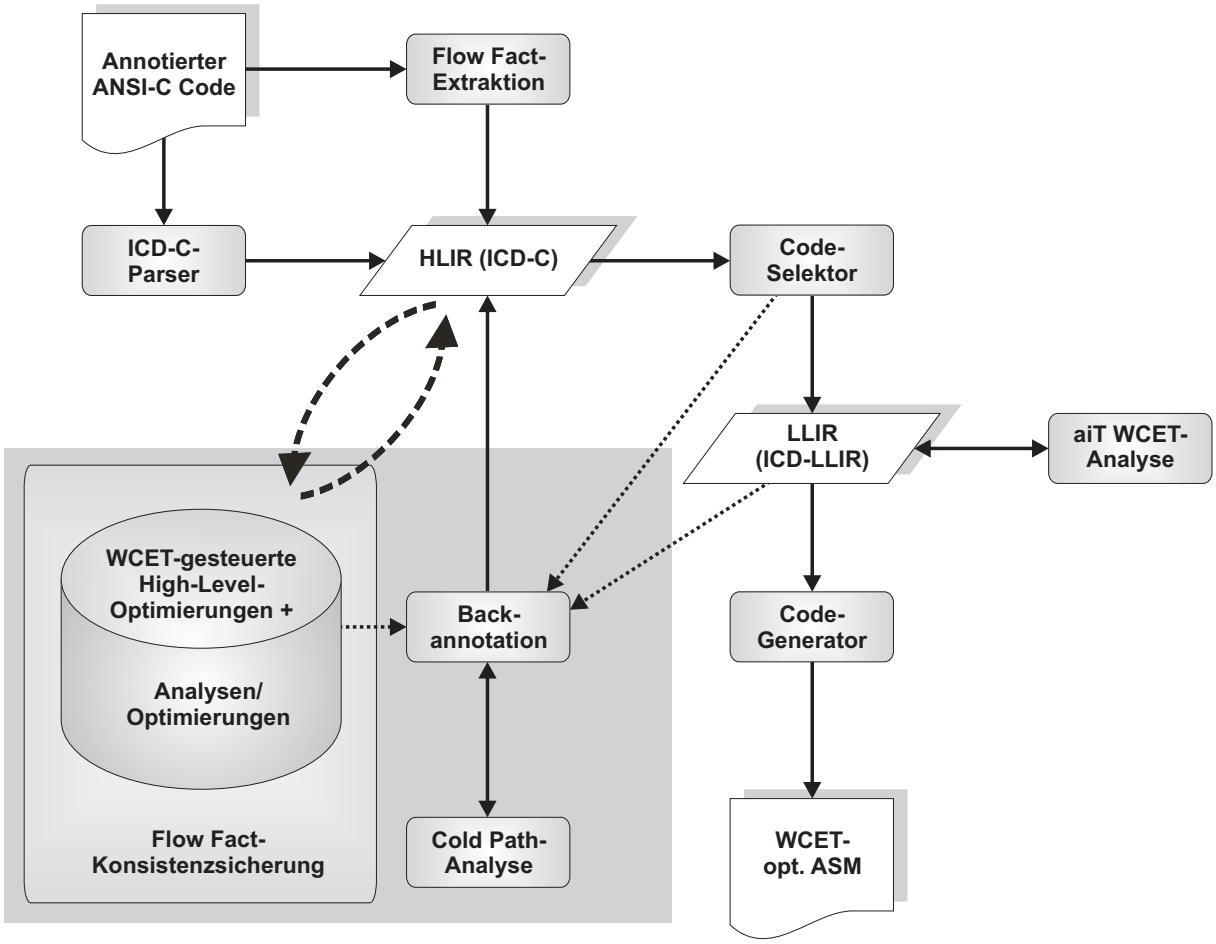


Abbildung 6.1: *WCC: Ausnutzung von WCET_{est}-Informationen durch High-Level-Optimierungen*

Dieses Kapitel beschäftigt sich mit der Entwicklung einer WCET-gesteuerten Heuristik für die High-Level-Optimierung **Function Inlining**. Für diesen Zweck wird das Verfahren Zufallswälder aus dem Bereich des maschinellen Lernens eingesetzt, um erstmals für den WCC automatisch eine Heuristik für eine Optimierung zu generieren. Die Heuristik soll die zentrale Frage beantworten, wann Function Inlining durchgeführt werden soll und wann nicht. Ziel ist die Entwicklung einer *möglichst allgemeinen* Heuristik, die im Idealfall für *alle* Benchmarks eine WCET-Reduzierung erreicht, auch wenn dadurch

die Heuristik konservativ wird, d.h. die Optimierung in bestimmten Situationen unterbunden wird, um im Schnitt eine WCET-Verbesserung zu erreichen.

In Abschnitt 6.1 wird zuerst die Optimierung Function Inlining näher vorgestellt. Dabei werden auch die Vor- und Nachteile dieser Optimierung betrachtet. Anschließend wird auf die Inlining-Heuristik der ICD-C eingegangen. Außerdem wird eine weitere Heuristik mit dem Namen **One-Call Function Inlining** vorgestellt, welche das Inlining nur für solchen Funktionen durchführt, die im gesamten Quellcode nur einen Funktionsaufruf besitzen. Abschnitt 6.2 bietet eine Einführung in das maschinelle Lernen. Dabei werden auch die Lernverfahren Entscheidungsbäume und Zufallswälder erklärt. Zufallswälder arbeiten auf der Basis vieler Entscheidungsbäume und versuchen dadurch das Ergebnis zu verbessern. Nachdem die Grundlagen für das maschinelle Lernen beschrieben wurden, werden in Abschnitt 6.3 verwandte Arbeiten diskutiert. In Abschnitt 6.4 wird erläutert, wie das Maschinelle Lernen einer WCET-gesteuerten Heuristik für die High-Level-Optimierung Function Inlining funktioniert.

6.1 Function Inlining

Function Inlining ist eine altbekannte Compiler-Optimierung. Die Idee besteht darin, dass ein Funktionsaufruf im Programm durch eine Kopie des Funktionsrumpfes der aufgerufenen Funktion ersetzt wird. Die aktuellen Parameter werden dabei in den kopierten Funktionsrumpf propagiert. Die Vor- und Nachteile der Optimierung werden an einem Beispiel erläutert. Abb. 6.2 zeigt die Situation vor dem Inlining der Funktion `bar`. Abb. 6.3 hingegen zeigt die Situation nach dem Inlining.

Durch das Inlining wird der Funktionsaufruf `bar( a, b, 7 )` in der Funktion `foo` durch den Funktionsrumpf von `bar` ersetzt. Die aktuellen Parameter `a`, `b` und `7` werden in den kopierten Funktionsrumpf von `bar` propagiert.

In der Regel erzeugt ein Compiler für jeden Funktionsaufruf einen neuen Bereich auf dem Stack. Dort werden neben den lokalen Variablen der aufgerufenen Funktion auch die Parameter der Funktion und der Rückgabewert abgelegt. Zudem müssen die nicht-flüchtigen Register eines Prozessors vor einem Funktionsaufruf gesichert werden, weil deren Inhalte auch über Funktionsgrenzen hinweg erhalten bleiben müssen. Da der Funktionsaufruf `bar( a, b, 7 )` nach dem Inlining entfernt wird, entfällt automatisch der Overhead für den Funktionsaufruf, was zu Laufzeiteinsparungen führt.

Ein weiterer großer Vorteil der Optimierung Function Inlining ist die Ermöglichung weiterer Standard-Optimierungen. Viele Standard-Optimierungen sind sogenannte *intraprozedurale* Optimierungen, d.h. sie werden lokal in einer Funktion f ausgeführt und betrachten nicht die Funktionen, die f aufruft, oder von denen f aufgerufen wird. Das Gegenteil einer intraprozeduralen Optimierung ist eine *interprozedurale* Optimierung, wie z.B. die Optimierung Procedure Cloning (vgl. Abschnitt 3.1.1.4). Dabei werden auch die Aufruf-Relationen zwischen Funktionen bei der Durchführung der Optimierung berücksichtigt. Durch die Optimierung Function Inlining wird mehr Optimierungspotential für Standard-Optimierungen geschaffen, weil der Funktionsrumpf der aufgerufenen Funktion in die aufrufende Funktion kopiert wird, sodass auch intraprozedurale Optimierungen, wie z.B. Constant Pro-

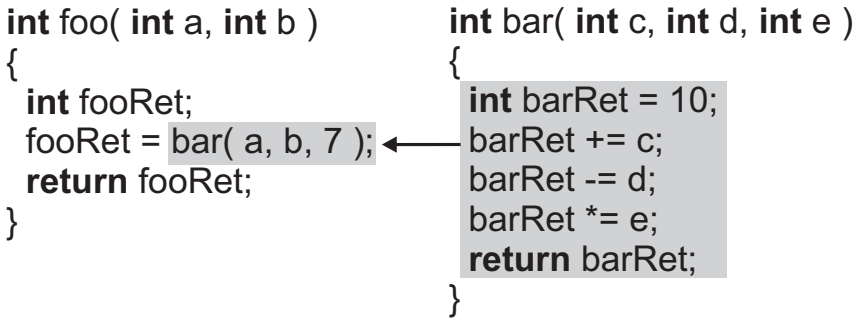


Abbildung 6.2: *Situation vor dem Inlining*

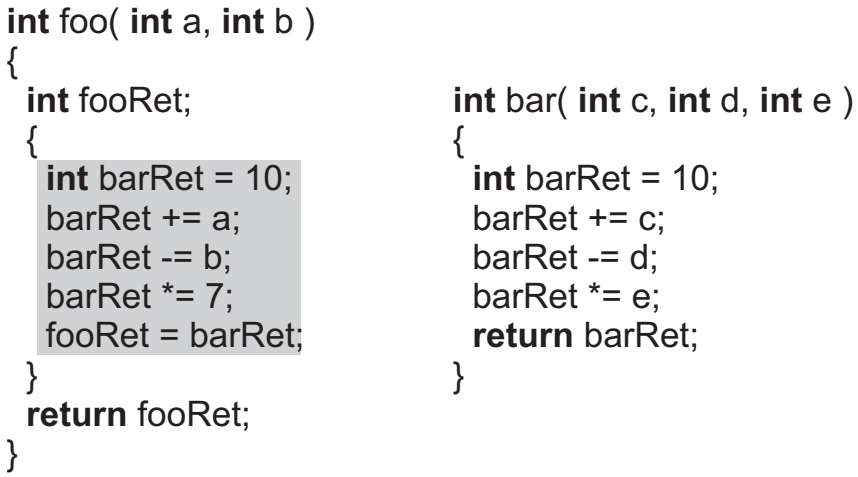


Abbildung 6.3: *Situation nach dem Inlining*

pagation, mehr Möglichkeiten zum Optimieren haben.

In Abb. 6.3 wird ein Nachteil der Optimierung Function Inlining sichtbar. Nach dem Inlining ist die Funktion `foo` größer geworden, die Funktion `bar` hat sich dagegen nicht verändert. Somit führt die Optimierung Function Inlining zu einer Code-Vergrößerung, falls die aufgerufene Funktion nicht aus dem Quellcode entfernt werden kann. Insbesondere bei Eingebetteten Systemen spielt die Code-Größe eine wichtige Rolle und ihr Zuwachs sollte minimal gehalten werden. Darüber hinaus kann eine Code-Vergrößerung zu einem Cache-Überlauf führen, was negative Auswirkungen auf die Programmlaufzeit hat.

Durch die Optimierung Function Inlining kann auch der Register-Druck zunehmen. So werden in Abb. 6.3 in der Funktion `foo` nach dem Inlining potentiell mehr Register benötigt, was zu Spill-Code führen kann. Auf das Thema wird in Abschnitt 6.4.1.1 näher eingegangen.

Folglich kann es in bestimmten Fälle nach der Durchführung der Optimierung Function Inlining zu einer Verschlechterung in der Programmlaufzeit kommen. Daher wird im Allgemeinen eine Heuristik eingesetzt, um zu entscheiden, wann Function Inlining durchgeführt werden soll und wann nicht.

6.1.1 ICD-C Function Inlining

Eine typische Heuristik, die auch in der ICD-C-Implementation der Optimierung Function Inlining zum Einsatz kommt, ist das Inlining von ausschließlich kleinen Funktionen. Innerhalb der ICD-C-Optimierung Function Inlining wird die Größe einer Funktion durch die Anzahl der in der Funktion enthaltenen ICD-C-Ausdrücke gemessen. In der ICD-C wird die Optimierung nur bei den Funktionen durchgeführt, die höchstens 50 Ausdrücke beinhalten und nicht rekursiv sind.

ICD-C Function Inlining wird im WCC durch Angabe der Option `-finline-functions` aufgerufen. Die Optimierung wird auf der höchsten Optimierungsstufe (O3) automatisch ausgeführt.

6.1.2 One-Call Function Inlining

Eine weitere bekannte Heuristik ist das Inlining von Funktionen, die im gesamten Quellcode des zu analysierenden Programms nur an einer Stelle aufgerufen werden. Solche Funktionen werden im Folgenden One-Call-Funktionen genannt. One-Call-Funktionen können auch mehrmals aufgerufen werden. Allerdings erfolgt jeder Funktionsaufruf immer von einer bestimmten Stelle aus.

Der große Vorteil bei One-Call-Funktionen besteht darin, dass das Inlining solcher Funktionen zu keiner Code-Vergrößerung führt: Zum Einen wird die ursprüngliche Funktion aus dem Quellcode entfernt, da sie nach dem Inlining nicht mehr gebraucht wird, und zum Anderen wird der zugehörige Funktionsaufruf entfernt, der bei vielen Parametern sehr lang sein kann.

6.2 Maschinelles Lernen

Das Problem bei der Optimierung Function Inlining besteht darin, zu entscheiden, ob das Inlining der aufgerufenen Funktionen eines Funktionsaufrufs sinnvoll ist oder nicht. Sinnvoll heißt in diesem Zusammenhang, dass nach dem Inlining eine Laufzeit-Verbesserung erreicht wird. In Abschnitt 6.1 wurde beschrieben, dass die Durchführung der Optimierung Function Inlining auch zu einer Verschlechterung in der Programmlaufzeit führen kann. Wie groß die Verschlechterung aufgrund der Optimierung Function Inlining sein kann, wird beispielhaft in Abb. 6.4 anhand von konkreten Benchmarks demonstriert.

Die Benchmarks in Abb. 6.4 wurden auf der höchsten Optimierungsstufe (O3) jeweils mit und ohne die ICD-C-Optimierung Function Inlining übersetzt. Im Schnitt führt die Optimierung Function Inlining zu einer erheblichen Laufzeit-Vergrößerung von 37,7%. Die Verschlechterung kann von vielen Faktoren abhängen, wie z.B. Speicher, Reihenfolge der Standard-Optimierungen oder Implementation der Optimierung Function Inlining. Ein Mensch kann in kurzer Zeit nicht alle Faktoren und deren Abhängigkeiten hinreichend genau analysieren. Zudem muss dieser Prozess nach jeder großen Änderung am Compiler erneut durchgeführt werden. Um diesen Prozess zu automatisieren, werden in dieser Arbeit Verfahren aus dem Bereich des maschinellen Lernens eingesetzt, um automatisch eine

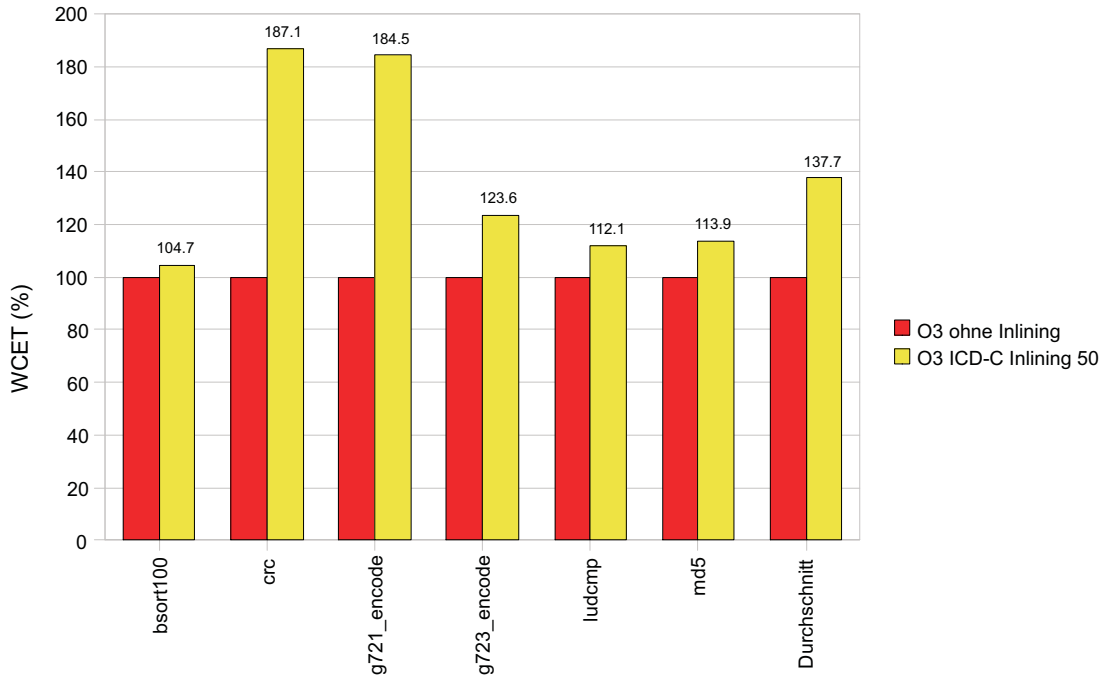


Abbildung 6.4: *Laufzeit-Vergrößerung aufgrund der Optimierung Function Inlining*

Heuristik für die Optimierung Function Inlining zu generieren. Eine gute Heuristik kann Situationen, die zu einer Verschlechterung in der Programmlaufzeit führen, frühzeitig erkennen (vorhersagen) und das Inlining in solchen Situationen unterbinden, sodass im Idealfall das Inlining nur dann durchgeführt wird, wenn auch eine Laufzeit-Verbesserung erreicht wird.

An dieser Stelle wird deutlich, dass es sich bei der Optimierung Function Inlining um ein *Klassifikationsproblem* mit zwei Klasen handelt: Inlining – *ja* oder *nein*. Die *Klassifikationsregel* (Heuristik) soll für jeden Funktionsaufruf c im zu optimierenden Programm P die korrekte Klassenzugehörigkeit ermitteln. Somit repräsentieren Funktionsaufrufe die zu klassifizierenden Objekte.

Eine Klassifikationsregel lässt sich mittels eines beliebigen *Klassifikationsverfahrens*, wie z.B. Entscheidungsbäume, Zufallswälder, Naive Bayes-Methode oder künstliche neuronale Netze, automatisch generieren. Das Klassifikationsverfahren erwartet als Eingabe einen *Trainingsdatensatz*, der Beispiele für bereits klassifizierte Funktionsaufrufe enthält. Neben einem Trainingsdatensatz gibt es auch einen *Testdatensatz* zur Beurteilung der Vorhersagegenauigkeit von Klassifikationsregeln. In Kapitel 8 wird das Testen von Klassifikationsregeln näher erläutert. Trainings- und Testdatensatz zusammen bilden den *Lerndatensatz*. Abb. 6.5 zeigt den Prozess der Konstruktion einer Klassifikationsregel. In Abb. 6.6 wird die Klassifikationsregel getestet. Abb. 6.7 zeigt schließlich die Anwendung der gelernten Klassifikationsregel auf eventuell unbekannte Daten.

Im Folgenden wird das Klassifikationsverfahren Entscheidungsbäume näher vorgestellt. Anschließend wird das Verfahren Zufallswälder präsentiert, das die Ergebnisse der Klassifikationen vieler Entscheidungsbäume zusammenfasst, um dadurch die Vorhersage zu verbessern.

Abbildung 6.5: *Prozess: Konstruktion einer Klassifikationsregel*

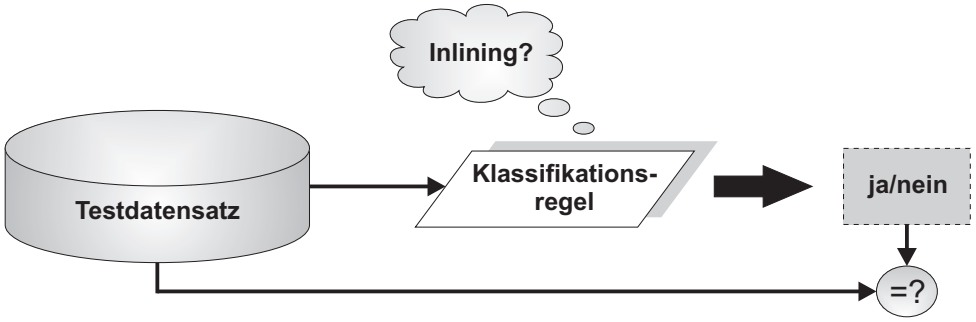


Abbildung 6.6: *Prozess: Testen einer Klassifikationsregel*

6.2.1 Entscheidungsbäume

Entscheidungsbäume dienen zur Klassifikation von Objekten, wie z.B. Funktionsaufrufe, und bestehen aus einer Abfolge von binären Entscheidungen, die als Baum darstellbar sind. Im Rahmen dieser Arbeit sind nur solche Entscheidungsbäume von Bedeutung, die eine binäre Klassifikation von Objekten zum Ziel haben. Das sind Entscheidungsbäume, die bei der Klassifikation von Objekten nur eine ja/nein-Entscheidung treffen müssen.

Der große Vorteil von Entscheidungsbäumen besteht darin, dass die Regeln der Entscheidungsbäume für den Anwender meist sehr verständlich sind. Ein Beispiel für einen Entscheidungsbaum zeigt Abb. 6.8. Der abgebildete Entscheidungsbaum soll die Entscheidung treffen, ob ein bestimmter Spielfilm gekauft werden soll oder nicht. In diesem Beispiel spiegelt der Entscheidungsbaum das Kaufverhalten eines bestimmten Kunden wieder. Der Entscheidungsbaum kann dazu genutzt werden, um automatisch Produktvorschläge für den Kunden zu erzeugen.

Die Wurzel und die inneren Knoten eines Entscheidungsbaumes bestehen aus Attributen. Ein Knoten repräsentiert eine Abfrage, um den Wert des betrachteten Objekts für das jeweilige Attribut zu ermitteln. Die möglichen Attributwerte stehen an den ausgehenden Kanten eines Knotens. Die Blätter stellen eine Klassifikation dar. Somit werden Objekte durch einen Entscheidungsbaum klassifiziert, indem durch Abfragen der vollständige Pfad von der Wurzel bis zum Blattknoten ermittelt wird [Kern-Isberner WS–2006/2007].

Im Folgenden wird erläutert, wie ein Entscheidungsbaum auf der Basis eines Trainingsdatensatzes, der bereits klassifizierte Objekte beinhaltet, konstruiert werden kann, sodass sowohl die Objekte aus dem Trainingsdatensatz als auch unbekannte Objekte nach Möglichkeit korrekt klassifiziert werden. In diesem Zusammenhang spricht man auch vom *induktiven Lernen*.

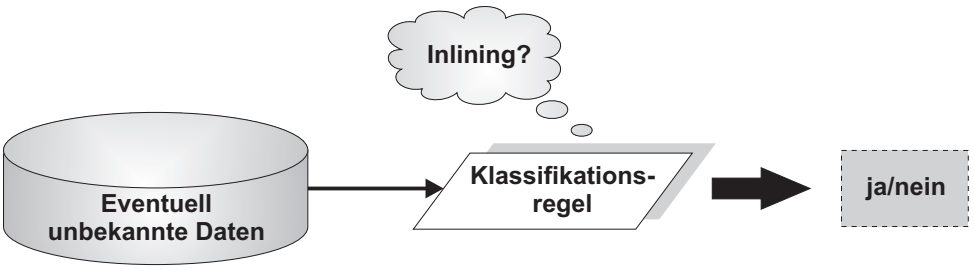


Abbildung 6.7: *Prozess: Anwendung einer Klassifikationsregel*

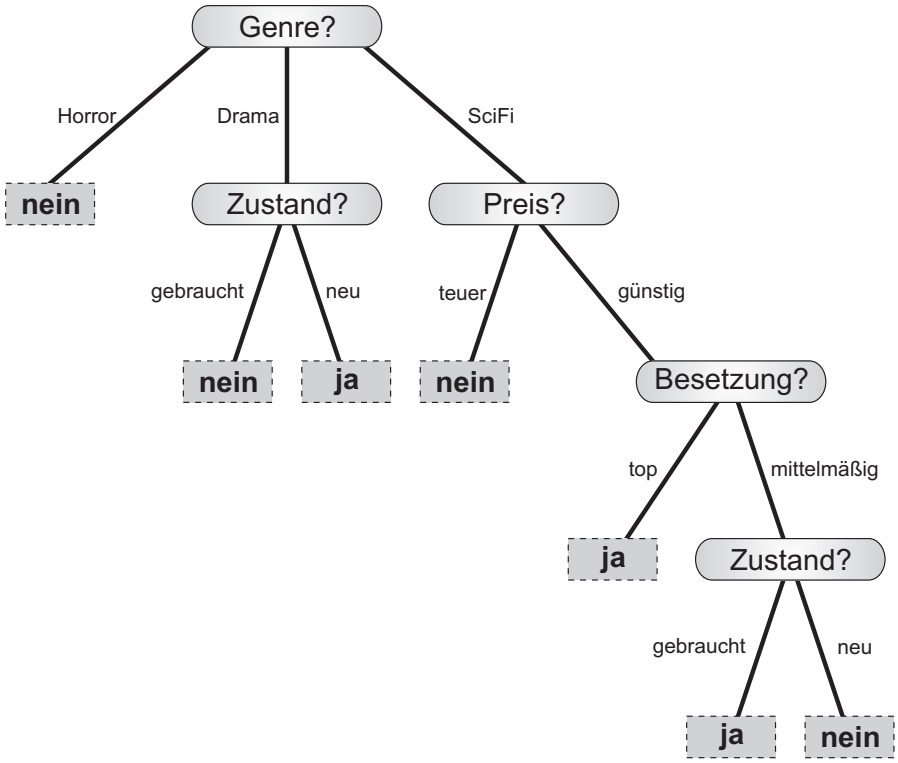


Abbildung 6.8: *Beispiel für einen Entscheidungsbaum*

6.2.1.1 Konstruktion von Entscheidungsbäumen

Nachfolgend wird der Algorithmus *Top-Down Induction of Decision Trees* (*TDIDT*) vorgestellt [Kern-Isberner u. Beierle 2006], mit dessen Hilfe ein Entscheidungsbaum zur binären Klassifikation von Objekten rekursiv aufgebaut werden kann.

- Zunächst wird ein Attribut a ausgewählt. Die Attributauswahl ist der wichtigste Schritt bei der Konstruktion von Entscheidungsbäumen. Weiter unten wird erläutert, mit welchen Verfahren das jeweils nächste Attribut für den Wuzelknoten und die inneren Knoten ausgewählt werden kann.
- Für jeden möglichen Attributwert von a wird ein Nachfolgeknoten erzeugt. Zudem werden die Kanten mit den Attributwerten markiert.
- Danach wird der aktuelle Trainingsdatensatz entsprechend den jeweiligen Attributwerten von

a auf die Nachfolgeknoten verteilt. Somit betrachten die Nachfolgeknoten nicht alle Beispiele aus dem aktuellen Trainingsdatensatz, sondern nur solche mit passenden Attributwerten.

- Anschließend wird der TDIDT-Algorithmus rekursiv für die neuen Nachfolgeknoten aus Schritt 2 ausgeführt ($\hookrightarrow$ Schritt 1).

Es gibt vier verschiedene Fälle, die an einem Nachfolgeknoten auftreten können.

- Wenn alle Beispiele im Nachfolgeknoten die gleiche Klassifikation haben, wird der Nachfolgeknoten mit dieser Klassifikation markiert. Im diesem Fall wurde ein Blattknoten im endgültigen Entscheidungsbaum erzeugt.
- Wenn die Beispielmenge im Nachfolgeknoten leer ist, wird der Nachfolgeknoten mit einer sinnvollen Standard-Klassifikation markiert. Die Beispielmenge im Nachfolgeknoten kann leer sein, wenn es z.B. im gesamten Trainingsdatensatz für bestimmte Attributwerte keine Beispiel gibt. In einer solchen Situation könnte z.B. das Inlining von Funktionen unterbunden werden (Standard-Klassifikation: *nein*), um zumindest keine Verschlechterung in der Programmlaufzeit zu verursachen.
- Wenn es noch positive und negative Beispiele im Nachfolgeknoten gibt, jedoch keine Attribute mehr übrig sind, dann ist der Trainingsdatensatz inkonsistent: Der Trainingsdatensatz enthält unterschiedlich klassifizierte Beispiele mit genau denselben Attributwerten. In diesem Fall kann z.B. eine Zuordnung zu der Klasse getroffen werden, die am häufigsten vorkommt.
- Wenn es noch positive und negative Beispiele im Nachfolgeknoten gibt und die aktuelle Menge der Attribute nicht leer ist, so wird der TDIDT-Algorithmus rekursiv für den aktuell betrachteten Nachfolgeknoten ausgeführt.

Um den vorgestellten Algorithmus besser verstehen zu können, wird im Folgenden anhand eines Beispiels gezeigt, wie ein Entscheidungsbaum auf der Basis eines gegebenen Trainingsdatensatzes aufgebaut werden kann. Der Entscheidungsbaum soll die Frage beantworten, ob ein bestimmter Spielfilm gekauft werden soll oder nicht. Somit repräsentieren Spielfilme die zu klassifizierenden Objekte. Tabelle 6.1 enthält relevante Attribute und die zugehörigen Attributwerte. Tabelle 6.2 zeigt den Trainingsdatensatz, der als Eingabe für das Klassifikationsverfahren Entscheidungsbäume dient.

Im ersten Schritt des TDIDT-Algorithmus wird zunächst ein Attribut ausgewählt. Bei der Auswahl wird das Attribut bevorzugt, das am *wichtigsten* ist. Es gibt unterschiedliche Kriterien zur Beurteilung der Wichtigkeit von Attributen, die im TDIDT-Algorithmus eingesetzt werden können. Nachfolgend werden die beiden bekannten Kriterien Kardinalitätskriterium und Information Gain-Kriterium vorgestellt [Kern-Isberner WS–2006/2007]. Dabei wird das Spielfilm-Problem als Beispiel benutzt.

6.2.1.2 Kardinalitätskriterium

Das Kardinalitätskriterium, das im Folgenden vorgestellt wird, wurde in dieser Arbeit nicht eingesetzt und dient einleitend zur Veranschaulichung der Vorgehensweise bei der Attributauswahl.

Attribut	Attributwerte
Genre	*Horror*, *Drama*, *SciFi*
Preis	*teuer*, *günstig*
Besetzung	*top*, *mittelmäßig*
Zustand	*neu*, *gebraucht*

Tabelle 6.1: *Relevante Attribute für das Spielfilm-Problem*

Nr.	Genre	Preis	Besetzung	Zustand	Klassifizierung
1	Horror	teuer	top	gebraucht	nein
2	Drama	günstig	mittelmäßig	neu	ja
3	SciFi	günstig	top	neu	ja
4	SciFi	günstig	mittelmäßig	neu	nein
5	Drama	teuer	top	neu	ja
6	SciFi	teuer	top	gebraucht	nein
7	SciFi	günstig	mittelmäßig	gebraucht	ja
8	Horror	günstig	mittelmäßig	gebraucht	nein
9	Horror	teuer	mittelmäßig	neu	nein
10	Drama	günstig	mittelmäßig	gebraucht	nein

Tabelle 6.2: *Trainingsdatensatz für das Spielfilm-Problem*

Nach dem Kardinalitätskriterium ist ein Attribut wichtig, wenn es soviele Beispiele wie möglich klassifiziert. Je mehr Beispiele durch das Attribut klassifiziert werden, desto wichtiger ist das Attribut.

Um beim Kardinalitätskriterium das (jeweils nächste) beste Attribut auswählen zu können, muss vorher ermittelt werden, wie viele Beispiele die in Frage kommenden Attribute vollständig klassifizieren können. Es wird zunächst das Attribut *Genre* betrachtet (vgl. Abb. 6.9).

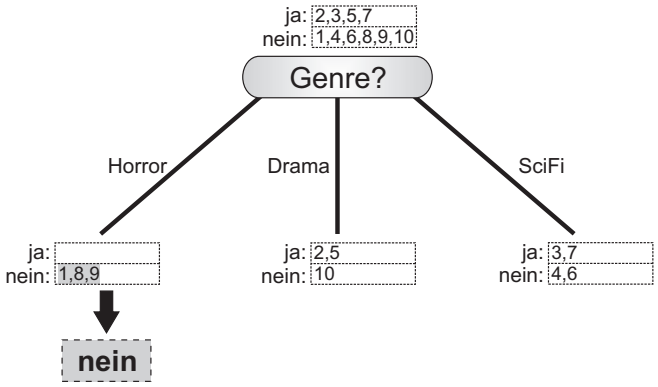


Abbildung 6.9: *Attributauswahl: Genre*

Die aktuelle Menge von positiven (2, 3, 5 und 7) und negativen Beispielen (1, 4, 6, 8, 9 und 10) wird entsprechend den möglichen Attributwerten von *Genre* verteilt. In Abb. 6.9 wird ersichtlich, dass beim Attributwert *Genre* = *Horror* drei Beispiele vollständig klassifiziert werden (1, 8 und 9). Die anderen Attributwerte von *Genre* können keine Beispiele eindeutig klassifizieren. Somit werden

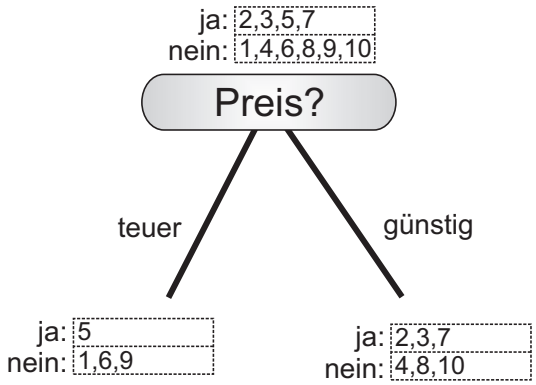


Abbildung 6.10: *Attributauswahl: Preis*

insgesamt drei Beispiele durch das Attribut *Genre* vollständig klassifiziert.

Danach wird für das Attribut *Preis* untersucht, wie viele Beispiele durch dieses Attribut klassifiziert werden können. Abb. 6.10 zeigt, dass kein einziges Beispiel mit nur einem Test vollständig klassifiziert werden kann.

Folglich ist das Attribut *Genre* nach dem Kardinalitätskriterium als erstes Attribut des Entscheidungsbaumes besser geeignet als das Attribut *Preis*. Die Attribute *Besetzung* und *Zustand* können ebenfalls kein einziges Beispiel vollständig klassifizieren. Daher wird im ersten Schritt des TDIDT-Algorithmus als erstes Attribut des Entscheidungsbaumes das Attribut *Genre* ausgewählt. Anschließend wird der TDIDT-Algorithmus auf die Nachfolgeknoten des neu erzeugten Wurzelknotens angewandt.

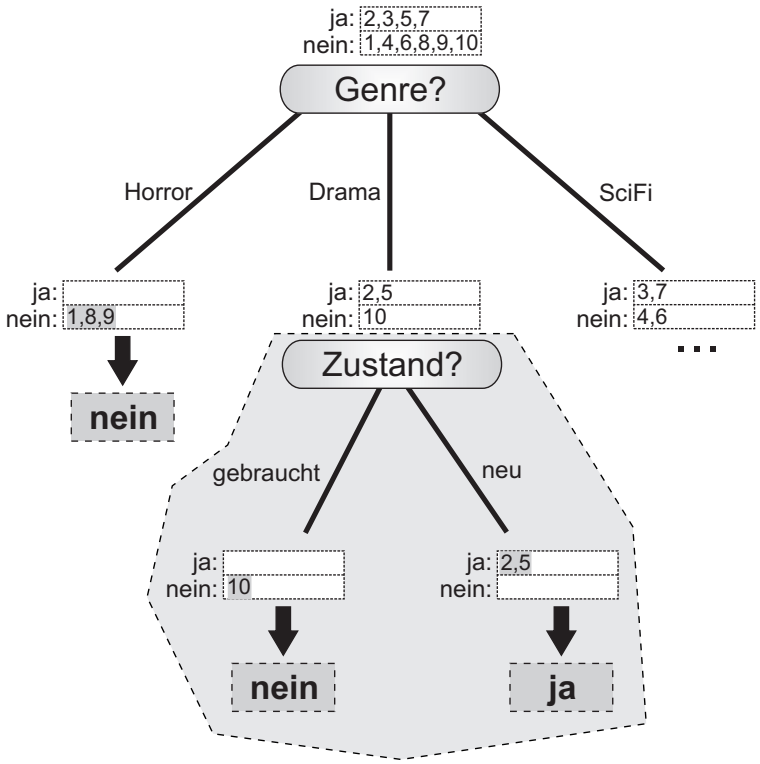


Abbildung 6.11: *TDIDT-Algorithmus wird rekursiv aufgerufen*

In Abb. 6.11 wird mittels des TDIDT-Algorithmus der mittlere Nachfolger des Wurzelknotens ermittelt. Dabei kommt das Attribut *Genre* nicht mehr in Frage, weil das Attribut bereits in einem Vorgängerknoten (im Wurzelknoten) enthalten ist. In der Abbildung wird ersichtlich, dass das Attribut *Zustand* alle Beispiele des aktuellen Trainingsdatensatzes (2, 5 und 10) vollständig klassifiziert. Somit wird dieses Attribut als nächstes Attribut ausgewählt. Der endgültige Entscheidungsbaum ist in Abb. 6.8 auf Seite 61 zu sehen.

6.2.1.3 Information Gain-Kriterium

Weil die etabliertesten TDIDT-Systeme zur Bestimmung des jeweils besten Attributs den Informationsgehalt eines Attributs verwenden, wird in dieser Arbeit das Information Gain-Kriterium benutzt. Beim Information Gain-Kriterium wird die Wichtigkeit eines Attributs a durch dessen Informationsgewinn $InfoGain(a)$ gemessen:

$$\boxed{InfoGain(a) = I(E) - I(E|a\ bekannt)} \tag{6.1}$$

$I(E)$ ist der Informationsgehalt der Beispielmenge E vor Auswahl eines Attributs und $I(E|a\ bekannt)$ ist der Informationsgehalt der Beispielmenge E nach Auswahl des Attributs a. Als Maß für den Informationsgehalt wird die Entropie benutzt. Der Begriff Entropie stammt aus der Thermodynamik und wurde später als Maß für die Informativität (Strukturiertheit) erkannt. Die Entropie bezeichnet den mittleren Informationsgehalt $H(P)$ einer Wahrscheinlichkeitsverteilung $P = (p_1, \ldots, p_n)$ und wird wie folgt berechnet:

$$H(P) = -\sum_{i=1}^{n} p_i * \log_2 p_i \tag{6.2}$$

Da in dieser Arbeit nur eine binäre Klassifikation von Objekten durchgeführt werden soll, gibt es in der Beispielmenge nur positive und negative Beispiele. Zudem wird angenommen, dass die Auswahl eines beliebigen Beispiels aus der Beispielmenge gemäß einer Gleichverteilung erfolgt. Bei p positiven und n negativen Beispielen beträgt die Wahrscheinlichkeit, ein positives bzw. negatives auszuwählen $\frac{p}{p+n}$ bzw. $\frac{n}{p+n}$. Der Informationsgehalt der Antwort auf die Frage, ob es sich bei einem beliebigen Beispiel um ein positives oder negatives Beispiel handelt, wird wie folgt berechnet:

$$I(E) := H(\frac{p}{p+n}; \frac{n}{p+n}) = -\frac{p}{p+n} * \log_2 \frac{p}{p+n} - \frac{n}{p+n} * \log_2 \frac{n}{p+n}\ bit \tag{6.3}$$

Der Informationsgehalt wird in bit gemessen, wobei ein bit dem Informationsgehalt einer ja/nein-Antwort entspricht. Falls die Entropie gleich null ist, so gehören alle Beispiele aus E zu derselben Klasse. Es kann leicht gezeigt werden, dass die Entropie der Gleichverteilung maximal ist. Somit ist die Beispielmenge E maximal unbestimmt, wenn sie genauso viele positive wie negative Beispiele besitzt.

Bei der Berechnung des Informationsgehalts einer Beispielmenge E nach Auswahl des Attributs a ($I(E|a\ bekannt)$) müssen alle möglichen Attributwerte von a berücksichtigt werden. Falls k die Anzahl der verschiedenen Attributwerte $w_1, \ldots, w_k$ ist, so wird die Beispielmenge E durch das Attribut

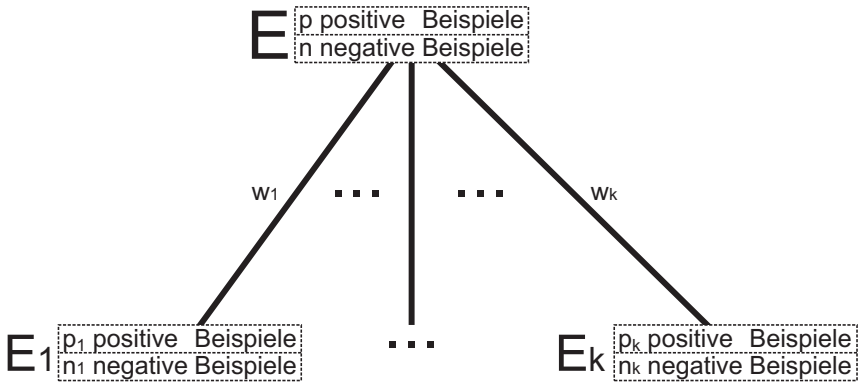


Abbildung 6.12: *Partitionierung der Beispielmenge E in disjunkte Teilmengen $E_1, \ldots, E_k$*

a in disjunkte Teilmengen $E_1, \ldots, E_k$ aufgeteilt. Die Partitionierung der Beispielmenge in disjunkte Teilmengen wird in Abb. 6.12 veranschaulicht. Somit gilt:

$$I(E|a\ bekannt) = \sum_{i=1}^{k} P(a = w_i) * I(E_i) = \sum_{i=1}^{k} \frac{p_i + n_i}{p + n} * H(\frac{p_i}{p_i + n_i}; \frac{n_i}{p_i + n_i})\ bit \tag{6.4}$$

Zusammengefasst lässt sich sagen, dass beim Information Gain-Kriterium das Attribut a als nächstes ausgewählt wird, bei dem der Informationsgewinn $InfoGain(a)$ maximal ist. Nähere Informationen sind in [Kern-Isberner u. Beierle 2006] enthalten.

Im Folgenden wird beispielhaft der Informationsgewinn des Attributs *Genre* ausgerechnet. Die Beispielmenge $E = 1, \ldots, 10$ besteht aus allen Beispielen des Trainingsdatensatzes für das Spielfilm-Problem (vgl. Tabelle 6.2 auf Seite 63).

$$I(E) = H(\frac{p}{p+n}; \frac{n}{p+n}) = H(\frac{4}{10}; \frac{6}{10}) = H(\frac{2}{5}; \frac{3}{5}) = -\frac{2}{5} * \log_2 \frac{2}{5} - \frac{3}{5} * \log_2 \frac{3}{5} \approx 0,97\ bit \tag{6.5}$$

$$\begin{aligned}
I(E|Genre\ bekannt) &= P(Genre = Horror) * I(\{1, 8, 9\}) \\
&+ P(Genre = Drama) * I(\{2, 5, 10\}) \\
&+ P(Genre = SciFi) * I(\{3, 4, 6, 7\}) \\
&= \frac{3}{10} * I(\{1, 8, 9\}) \\
&+ \frac{3}{10} * I(\{2, 5, 10\}) \\
&+ \frac{4}{10} * I(\{3, 4, 6, 7\}) \\
&= \frac{3}{10} * (-\frac{3}{3} * \log_2 \frac{3}{3}) \\
&+ \frac{3}{10} * (-\frac{2}{3} * \log_2 \frac{2}{3} - \frac{1}{3} * \log_2 \frac{1}{3}) \\
&+ \frac{4}{10} * (-\frac{2}{4} * \log_2 \frac{2}{4} - \frac{2}{4} * \log_2 \frac{2}{4}) \\
&\approx 0,68\ bit
\end{aligned} \tag{6.6}$$

$$InfoGain(Genre) = 0,97\ bit - 0,68\ bit = 0,29\ bit \tag{6.7}$$

Somit beträgt der Informationsgewinn des Attributs *Genre* $0,29\ bit$. Der Informationsgewinn aller anderen Attribute ist geringer als der Informationsgewinn des Attributs *Genre*. Daher wird auch beim Information Gain-Kriterium als erstes Attribut *Genre* ausgewählt. Der am Ende resultierende Entscheidungsbaum unterscheidet sich in diesem Beispiel nicht von dem Entscheidungsbaum, der mittels des Kardinalitätskriteriums aufgebaut wurde (vgl. Abb. 6.8 auf Seite 61).

6.2.2 Zufallswälder

Zufallswälder (Random Forests) beschreiben ein weiteres Klassifikationsverfahren [Breiman 2001]. Sie werden in dieser Arbeit benutzt, um eine Verbesserung der Vorhersage zu erreichen. Zufallswälder klassifizieren einzelne Objekte auf der Basis vieler Entscheidungsbäume. Die Konstruktion der Entscheidungsbäume ist dabei zufallsabhängig. Aus diesem Grund wurde der Name Zufallswälder gewählt.

Entscheidungsbäume und Zufallswälder sind Verfahren aus dem Bereich des *überwachten* Lernens. Beim überwachten Lernen ist für jedes Beispiel aus dem Trainingsdatensatz die Klassenzugehörigkeit bekannt. Somit werden die Klassen durch den Benutzer vorgegeben. Beim *unüberwachten* Lernen muss das Verfahren selbst die Klassen ermitteln (Clustering).

6.2.2.1 Konstruktion

Am Anfang muss festgelegt werden, aus wie vielen Entscheidungsbäumen der Zufallswald bestehen soll. Die Klassifikationsgüte eines Zufallswaldes wird meistens umso besser, je mehr einzelne Entscheidungsbäume berechnet werden. Zudem führt eine große Anzahl an Entscheidungsbäumen nicht zu *Overfitting* und ist deshalb zu bevorzugen [Morik u. Ligges SS–2007]. Overfitting muss vermieden werden, weil beim Overfitting eine Überanpassung der gelernten Klassifikationsregel an den Trainingsdatensatz erfolgt, sodass die Generalisierungseigenschaft der Klassifikationsregel verloren geht.

Bei der Konstruktion eines jeden einzelnen Entscheidungsbaumes wird wie folgt vorgegangen: Der Entscheidungsbaum wird nicht aus allen zur Verfügung stehenden Trainingsdaten bestimmt. Stattdessen wird für jeden einzelnen Entscheidungsbaum eine Stichprobe aus dem Trainingsdatensatz gezogen. Dabei kann es auch passieren, dass ein Beispiel aus dem Trainingsdatensatz mehrmals gezogen wird.

Außerdem werden bei der Konstruktion eines Entscheidungsbaumes nicht alle Attribute verwendet. Stattdessen werden die Attribute, die bei der Konstruktion des Entscheidungsbaumes berücksichtigt werden sollen, zufällig bestimmt.

Zudem wird bei der Konstruktion eines Entscheidungsbaumes das Stutzen (Pruning) der Äste des

Baumes nicht verwendet, um möglichst große Bäume zu erhalten, die den kleinstmöglichen Wiedereinsetzungsfehler aufweisen.

6.2.2.2 Klassifikation

Ein Objekt wird durch einen Zufallswald klassifiziert, indem es durch jeden einzelnen Entscheidungsbaum im Zufallswald klassifiziert wird. Anschließend wird das Objekt der Klasse zugeordnet, die von den meisten Entscheidungsbäumen im Zufallswald bevorzugt wurde.

Der Vorteil eines Zufallswaldes gegenüber einem einzigen Entscheidungsbaum besteht darin, dass auch Attribute, die nur einen geringen Beitrag zur Klassentrennung liefern, irgendwann auch bei der Klassifikation verwendet werden, wohingegen solche Attribute in einem einzigen Entscheidungsbaum unberücksichtigt bleiben können, weil ein einziger Entscheidungsbaum alle Attribute verwenden darf und logischerweise die "wichtigsten" bevorzugt. Die Zusammenfassung der Ergebnisse der Klassifikationen vieler Entscheidungsbäume führt in der Regel zu einer Verbesserung der Vorhersage. Zudem liefern Zufallswälder als Nebenerzeugnis eine Einschätzung der Wichtigkeit der verwendeten Attribute (Variable Importance Measure). Die Wichtigkeit eines Attributs kann durch den Beitrag des Attributs zur Klassentrennung bestimmt werden. In Kapitel 8 wird das Verfahren näher erläutert.

Der Nachteil eines Zufallswaldes gegenüber einem einzigen Entscheidungsbaum besteht darin, dass die Verständlichkeit verloren geht, da die Klassifikationsregel nicht mehr einfach ablesbar ist. Zudem erhöht sich die Komplexität der Klassifikation, weil mehr Bäume zur Klassifikation eines Objekts benutzt werden.

Aufgrund der beschriebenen Vorteile wird im Rahmen dieser Diplomarbeit das Klassifikationsverfahren Zufallswälder eingesetzt, um eine WCET-gesteuerte Heuristik für die Optimierung Function Inlining zu generieren. Dabei werden die Entscheidungbäume im Zufallswald nach dem Information Gain-Kriterium aufgebaut.

6.3 Verwandte Arbeiten

Mit dieser Arbeit wird erstmalig ein maschinelles Lernverfahren eingesetzt, um eine Heuristik für eine High-Level-Optimierung zu generieren, die eine WCET-Minimierung zum Ziel hat. Die Arbeiten, die im Folgenden vorgestellt werden, versuchen die durchschnittliche Laufzeit von Programmen (ACET) zu verbessern.

In [Monsifrot u.a. 2002] wird das Klassifikationsverfahren Entscheidungsbäume benutzt, um automatisch eine Heuristik für die Optimierung Loop Unrolling zu generieren. Die generierte Heuristik soll entscheiden, ob eine bestimmte Schleife abgerollt werden soll oder nicht. Um Schleifen möglichst gut beschreiben zu können, werden zunächst potentiell wichtige Attribute eingeführt, wie z.B. Größe der Schleife oder Anzahl der arithmetischen Operationen in der Schleife. Anschließend wird der Lern-

datensatz aufgebaut. Für diesen Zweck werden 1036 Schleifen in 77 Benchmarks betrachtet. Dabei wird festgehalten, ob das Abrollen einer Schleife gewinnbringend ist oder nicht. Dies wird durch Simulationen festgestellt. Schleifen, die eine sehr geringe Ausführungszeit haben, werden im Lerndatensatz nicht berücksichtig, weil sie nicht repräsentativ genug sind. Außerdem werden Beispiele mit gleichen Attributwerten und möglicherweise unterschiedlichen Klassifikationen zusammengefasst. Am Ende umfasst der Lerndatensatz nur noch 572 Beispiele (139 positive und 433 negative Beispiele). Um die Genauigkeit der Vorhersage durch Entscheidungsbäume zu verbessern, wird das Verfahren *Boosting* benutzt. Beim Boosting wird ähnlich wie beim Klassifikationsverfahren Zufallswälder ein bestimmtes Klassifikationsverfahren mehrfach angewandt, allerdings nur rekursiv auf dem Teil der Daten, die besonders schwierig zu klassifizieren sind. Anschließend wird die Genauigkeit der gewonnenen Heuristik mittels der Leave-One-Out Cross-Validation-Methode getestet. Diese Methode wird in Kapitel 8 näher erläutert.

Auch in [Stephenson u. Amarasinghe 2005] werden maschinelle Lernverfahren eingesetzt, um die Heuristik für die Optimierung Loop Unrolling zu verbessern. Anders als in [Monsifrot u.a. 2002] versuchen die Autoren mit Hilfe von maschinellen Lernverfahren einen optimalen Abroll-Faktor vorherzusagen. Jeder Abroll-Faktor wird durch eine eigene Klasse repräsentiert. Weil es mehr als zwei Klassen gibt, wird eine multiple Klassifikation durchgeführt, wohingegen in [Monsifrot u.a. 2002] nur eine binäre Klassifikation durchgeführt wird. Insgesamt werden mehr als 2500 Schleifen aus 72 Benchmarks betrachtet, um einen Lerndatensatz aufzubauen. Die eingesetzten Klassifikationsverfahren sind Support Vector Machines (SVM) und Near Neighbor (NN). Auch in dieser Arbeit wird die Leave-One-Out Cross-Validation-Methode eingesetzt, um die Genauigkeit der Vorhersage durch SVM und NN einzuschätzen.

Im Gegensatz zu den vorgestellten Arbeiten, die sich mit der Optimierung Loop Unrolling beschäftigen, wird in [Cavazos u. O'Boyle 2005] die Optimierung Function Inlining betrachtet. Ein genetischer Algorithmus wird eingesetzt, um automatisch die Heuristik für das Inlining zu verbessern. Der Algorithmus versucht eine möglichst gute Belegung der Parameter zu finden, mit denen die Optimierung Function Inlining beeinflusst und gesteuert werden kann.

6.4 Maschinelles Lernen einer WCET-gesteuerten Function Inlining-Heuristik

Die manuelle Entwicklung bzw. Anpassung von Heuristiken ist mit einem erheblichen kognitiven Aufwand verknüpft. Durch das maschinelle Lernen einer Heuristik soll dem Entwickler diese mühsame Arbeit abgenommen werden.

In diesem Abschnitt wird gezeigt, wie die vorgestellten Konzepte aus dem Bereich des maschinellen Lernens für den WCC eingesetzt werden können, um erstmals für den WCC automatisch eine Heuristik für eine Optimierung zu *generieren*. Zudem wird in dieser Arbeit zum ersten Mal ein maschinelles Lernverfahren eingesetzt, um eine WCET-Minimierung zu erreichen.

Zunächst werden in Abschnitt 6.4.1 potentiell wichtige Attribute für die Optimierung Function Inlining vorgestellt, welche den Funktionsaufruf näher charakterisieren sollen. Außerdem wird der Register Pressure Analyzer vorgestellt, der vier weitere Attribute einführt, mit deren Hilfe Aussagen über den Register-Druck bereits vor dem Inlining einer Funktion gemacht werden können. Anschließend wird in Abschnitt 6.4.2 ein Framework für das maschinelle Lernen präsentiert, das ein manuelles Inlining-System beinhaltet. Das Framework ermöglicht den *automatischen* Aufbau eines Lerndatensatzes. Nachdem der obligatorische Lerndatensatz aufgebaut wurde, werden in Abschnitt 6.4.3 die für das maschinelle Lernen eingesetzten Programme RapidMiner und GNU R vorgestellt. Am Ende wird beschrieben, wie die gelernten Klassifikationsregeln, die WCET-gesteuerte Heuristiken für die Optimierung Function Inlining darstellen, implementiert und im WCC eingesetzt werden können.

6.4.1 Auswahl an Attributen für die Optimierung Function Inlining

Im Folgenden werden numerische und nominale Attribute vorgestellt, die für die Optimierung Function Inlining von Bedeutung sein könnten. Die Attributauswahl ist eine schwierige Aufgabe und in der wissenschaftlichen Literatur finden sich lediglich erste rudimentäre Ansätze. Im Folgenden werden solche Attribute ausgewählt, die eine möglichst gute Klassifikation versprechen. Dabei werden auch WCET$_{est}$-Informationen, die nach der Back-annotation-Phase auf High-Level IR-Ebene verfügbar sind, ausgenutzt. Diese Attribute werden später beim maschinellen Lernen einer WCET-gesteuerten Heuristik für die Optimierung Function Inlining zum Einsatz kommen, da sie eine Charakterisierung von Funktionsaufrufen ermöglichen.

Zuerst werden numerische Attribute vorgestellt, deren Wertebereich meistens der Menge der natürlichen Zahlen $\mathbb{N}_0$ entspricht.

- Größe der aufgerufenen Funktion
- Größe der aufrufenden Funktion

  Die Größe einer Funktion wird gemessen in Anzahl ICD-C-Ausdrücke.

- WCET$_{est}$ der aufgerufenen Funktion
- WCET$_{est}$ der aufrufenden Funktion
- Anzahl der Ausführungen des ersten IR-Basis-Blocks der aufgerufenen Funktion
- Anzahl der Ausführungen des ersten IR-Basis-Blocks der aufrufenden Funktion
- Anzahl der Ausführungen des IR-Basis-Blocks, der den Funktionsaufruf (Inline-Kandidat) enthält

  WCET$_{est}$-Informationen sind nach der Back-annotation-Phase verfügbar (siehe Kapitel 4) und werden an dieser Stelle ausgenutzt.

- Geschätzte verursachte WCET$_{est}$ des Funktionsaufrufs

  Die geschätzte verursachte WCET$_{est}$ des Funktionsaufrufs wird wie folgt berechnet:

  $$\lfloor a * \frac{b}{c} \rfloor, \text{ wobei}$$

  $a :=$Anzahl der Ausführungen des IR-Basis-Blocks, der den Funktionsaufruf enthält,

b :=Akkumulierte WCET$_{est}$ der aufgerufenen Funktion über alle Kontexte und
c :=Anzahl der Ausführungen des ersten IR-Basis-Blocks der aufgerufenen Funktion.

- Relative WCET$_{est}$ der aufgerufenen Funktion
- Relative WCET$_{est}$ der aufrufenden Funktion
- Relative geschätzte verursachte WCET$_{est}$ des Funktionsaufrufs

  Der prozentuale Anteil der WCET$_{est}$ an der globalen WCET.

- Statische Anzahl der Funktionsaufrufe in der aufgerufenen Funktion
- Statische Anzahl der Funktionsaufrufe in der aufrufenden Funktion

  Diese Attribute stehen für die Anzahl der Stellen im Quellcode der Funktion, die einen Funktionsaufruf beinhalten.

```
1  void foo()
2  {
3    int i;
4    bar();
5    for ( i = 0; i < 10; ++i ) {
6      foobar();
7    }
8    bar();
9  }
```

Abbildung 6.13: *Beispiel für die Ermittlung der statischen Anzahl der Funktionsaufrufe*

So beträgt z.B. in Abb. 6.13 die statische Anzahl der Funktionsaufrufe in der Funktion `foo` drei: `bar()` (Zeile 4), `foobar()` (Zeile 6) und `bar()` (Zeile 8).

- Statische Anzahl der Funktionsaufrufe zu der aufgerufenen Funktion

  Die Anzahl der Stellen im Quellcode des zu analysierenden Programms, die einen Funktionsaufruf zu der aufgerufenen Funktion beinhalten.

- Anzahl der Funktionsaufrufe zu der aufgerufenen Funktion, die zwangsläufig ausgeführt werden (vgl. Abschnitt 5.2) und auf dem WCEP liegen
- Anzahl der Funktionsaufrufe zu der aufgerufenen Funktion, die zwangsläufig ausgeführt werden und auf dem Cold Path liegen (siehe Kapitel 5)

Im Folgenden werden nominale Attribute mit den möglichen Werten {JA, NEIN} vorgestellt.

- One-Call-Funktion
  Dieses Attribut dient zum Kennzeichnen von One-Call-Funktionen.
- Funktionsaufruf in Schleife
  Dieses Attribut dient zur Unterscheidung von Funktionsaufrufen, die in Schleifen plaziert sind.

Aus den Cold Path Ergebnissen aus Kapitel 5 kann man folgern, dass die meisten Funktionen auf dem Cold Path und damit auch auf dem WCEP liegen. Diese Informationen sind beim maschinellen

Lernen einer Heuristik für die Optimierung Function Inlining weniger interessant, weil sie zu einer Trennung von Klassen wenig beitragen würden.

6.4.1.1 Register Pressure Analyzer

Function Inlining ist eine Optimierung, die den Register-Druck erhöhen kann. Wenn in einer Funktion ein Funktionsaufruf stattfindet, werden die nicht-flüchtigen Register automatisch gesichert, damit die aufgerufene Funktion auf freie Register zugreifen kann. Falls aber Function Inlining durchgeführt wird, wird der Funktionsaufruf durch den Funktionsrumpf der aufgerufenen Funktion ersetzt. Aus diesem Grund werden auch keine Register gesichert, sodass nach dem Inlining für den Funktionsrumpf der aufgerufenen Funktion weniger freie Register zur Verfügung stehen als vor dem Inlining. Der Register-Druck nimmt zu.

Ein erhöhter Register-Druck führt in der Regel zu Spilling. In Abschnitt 3.1.5 wurde erwähnt, dass Spill-Code negative Auswirkungen auf die Programmlaufzeit hat. Daher sollte im Idealfall eine WCET-gesteuerte Heuristik für die Optimierung Function Inlining Spill-Code vor dem Inlining einer Funktion vorhersagen können, um dadurch eine WCET-Verschlechterung aufgrund des Spill-Codes zu verhindern. Der Register Pressure Analyzer (RPA) wurde entwickelt, um für die WCET-gesteuerte Heuristik Informationen bereitzustellen, die als Indiz für Spill-Code verwendet werden können.

Der RPA betrachtet nur die Register, die potentiell gespillt werden können. Im WCC sind das die Daten- und Adressregister, die zum sogenannten oberen Kontext (Upper Context) gehören. Als Eingabe erhält der RPA die LLIR eines Programms. Dabei handelt es sich um die physikalische LLIR, für die bereits eine Register-Allokation durchgeführt wurde (vgl. Abschnitt 3.1.4). Anschließend ermittelt der RPA für jeden Funktionsaufruf im Programm die folgenden Werte:

- Anzahl der Live-Out Daten-Register des Funktionsaufrufs
- Anzahl der Live-Out Adress-Register des Funktionsaufrufs

Die Live-Out Register eines Funktionsaufrufs sind Register, die auch nach dem Funktionsaufruf gebraucht werden, d.h. die Lebenszeit der Register erstreckt sich über den Funktionsaufruf. Falls die Anzahl der Live-Out Daten- und Adress-Register eines Funktionsaufrufs relativ groß ist, dann ist auch die Wahrscheinlichkeit groß, dass nach dem Inlining Spill-Code erzeugt wird.

Außerdem ermittelt der RPA für jede Funktion die folgenden Werte:

- Maximale Anzahl an Daten-Registern, die gleichzeitig in der Funktion lebendig sind
- Maximale Anzahl an Adress-Registern, die gleichzeitig in der Funktion lebendig sind

Falls z.B. die Anzahl der Live-Out Daten- und Adress-Register eines Funktionsaufrufs relativ gering ist, dann kann es nach dem Function Inlining immer noch zu Spilling kommen, wenn in der aufgerufenen Funktion die Anzahl der gleichzeitig gebrauchten Register relativ groß ist. Daher werden diese beiden Attribute nur für die aufgerufene Funktion betrachtet.

Zusammengefasst lässt sich sagen, dass der RPA für den Lerndatensatz vier zusätzliche numerische Attribute zur Verfügung stellt. Es wird erhofft, dass mit Hilfe dieser Attribute eine bessere Klassifikation von Funktionsaufrufen ermöglicht wird.

6.4.2 Konstruktion eines Frameworks für das maschinelle Lernen

6.4.2.1 Manuelles Inlining-System

In [Cooper u.a. 1991] wird ein manuelles Inlining-System für Fortran-Programme vorgestellt. Der Benutzer kann Funktionsaufrufe, für die Function Inlining durchgeführt werden soll, manuell markieren. Anschließend wird (nur) für die markierten Funktionsaufrufe Function Inlining durchgeführt. So können neue Inlining-Strategien schnell getestet werden.

Inspiriert von der Arbeit von [Cooper u.a. 1991] wurde in dieser Arbeit ein manuelles Inlining-System für C-Programme entwickelt, um den Prozess der Extraktion von Beispielen aus den Benchmarks zu vereinfachen. Das manuelle Inlining-System wurde in die WCET-gesteuerte Optimierung Function Inlining integriert. Die WCET-gesteuerte Optimierung Function Inlining wird im WCC durch Angabe der Option `-fWCET-inline-functions` aufgerufen. Die Optimierung kann sich, u.a. abhängig von einer bestimmten Text-Datei mit dem Namen `inline.txt`, in unterschiedlichen Zuständen befinden. Mit Hilfe der `inline.txt`-Datei kann der Benutzer selbst entscheiden, für welche Funktionsaufrufe Function Inlining durchgeführt werden soll. Abb. 6.14 zeigt den allgemeinen Aufbau der `inline.txt`-Datei.

```
1 Funktionsname(1)   Funktionsaufruf-Nummer(1) ... Funktionsaufruf-Nummer(m)
2       ...
3       ...
4       ...
5 Funktionsname(N)   Funktionsaufruf-Nummer(1) ... Funktionsaufruf-Nummer(n)
```

Abbildung 6.14: *Allgemeiner Aufbau der `inline.txt`-Datei*

In Abb. 6.14 ist zu sehen, dass die Funktionsaufrufe einer Funktion, für die Function Inlining durchgeführt werden soll, in einer Zeile stehen. Der Funktionsname steht am Anfang einer jeden Zeile. Dahinter stehen die Funktionsaufrufe, die durch eine Nummer kodiert sind. Die Nummer i entspricht z.B. dem i-ten Funktionsaufruf einer bestimmten Funktion relativ zu seiner Position im Quellcode. Dabei wird die Reihenfolge der Funktionsaufrufe einer bestimmten Funktion durch die ICD-C vorgegeben. Üblicherweise betrachtet die ICD-C die Funktionsaufrufe in einer Übersetzungseinheit von oben nach unten und von links nach rechts bzw. bei verschachtelten Funktionsaufrufen von innen nach außen. Bei mehreren Übersetzungseinheiten muss zusätzlich die Reihenfolge der Übersetzungseinheiten berücksichtigt werden. Abb. 6.15 zeigt ein Beispiel. Mit Hilfe der `inline.txt`-Datei werden die Funktionsaufrufe mit grauem Hintergrund ausgewählt.

Im Folgenden werden die wesentlichen Zustände der WCET-gesteuerten Function Inlining-Optimierung

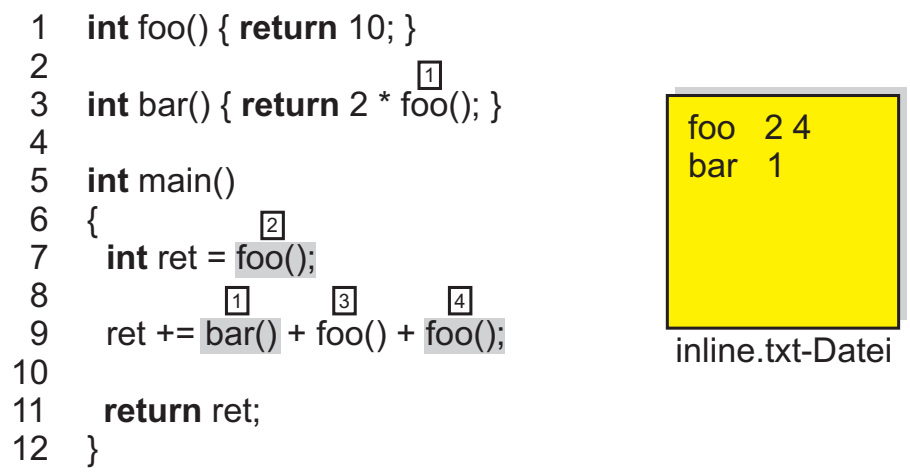


Abbildung 6.15: *Beispiel für eine `inline.txt`-Datei*

vorgestellt.

- **Zustand: Anwendung der WCET-gesteuerten Heuristik**
  Falls der Benutzer keine `inline.txt`-Datei bereitgestellt hat, befindet sich die Optimierung automatisch in dem Zustand der Anwendung einer WCET-gesteuerten Heuristik. In diesem Zustand wird die WCET-gesteuerte Heuristik für die Optimierung Function Inlining benutzt. Die Heuristik wird in den nachfolgenden Abschnitten entwickelt.
- **Zustand: Manuelles Inlining**
  Falls der Benutzer im Benchmark-Verzeichnis eine `inline.txt`-Datei bereitgestellt hat, wird die interne WCET-gesteuerte Heuristik deaktiviert. Stattdessen wird Function Inlining nur für die Funktionsaufrufe durchgeführt, die in der `inline.txt`-Datei angegeben sind.
- **Zustand: Lernen einer WCET-gesteuerten Heuristik**
  Falls `-fWCET-inline-functions` mit dem Parameter `learn-inline-functions` aufgerufen wird, so befindet sich die Optimierung in diesem Zustand. Auch in diesem Zustand wird die interne WCET-gesteuerte Heuristik deaktiviert. Wofür dieser Zustand gebraucht wird, wird im nächsten Abschnitt erklärt.

6.4.2.2 Aufbau des Lerndatensatzes

Der Aufbau eines Lerndatensatzes wurde im Rahmen dieser Diplomarbeit automatisiert. Für diesen Zweck wurde ein Bash-Skript geschrieben, welches den WCC benutzt, um Beispiele für den Lerndatensatz zu gewinnen. Nachfolgend wird die Funktionsweise erklärt.

Für jeden einzelnen Benchmark führt das Skript die folgenden Schritte aus:

1. Zuerst wird die Referenz-WCET$_{est}$ eines Benchmarks ermittelt. Für diesen Zweck wird der Benchmark mit der höchsten Optimierungsstufe (O3) übersetzt, wobei die ICD-C-Optimierung Function Inlining, die auf der höchsten Optimierungsstufe automatisch ausgeführt wird, deak-

tiviert wird:
`wcc -O3 -fno-inline-functions` *benchmark*

2. Anschließend wird der Benchmark erneut mit der höchsten Optimierungsstufe übersetzt. Diesmal wird jedoch `-fWCET-inline-functions` mit dem Parameter `learn-inline-functions` aufgerufen. Dadurch wird zum Einen das ICD-C Function Inlining deaktiviert und zum Anderen wird eine Datei mit dem Namen `learn_inline.txt` erzeugt, die für jeden Funktionsaufruf im Programm einen Einträg enthält. So wird z.B. für den i-ten Funktionsaufruf einer Funktion eine neue Zeile in der `learn_inline.txt`-Datei mit dem Inhalt „Funktionsname i“ erzeugt. Dabei werden nur solche Funktionsaufrufe betrachtet, die für das ICD-C Function Inlining in Frage kommen, und nicht aufgrund von ICD-C-Restriktionen ausgeschlossen wurden. So werden z.B. keine Funktionsaufrufe zu rekursiven Funktionen betrachtet.
`wcc -O3 -fWCET-inline-functions --param learn-inline-functions` *benchmark*
3. Für jede Zeile z der `learn_inline.txt`-Datei werden die folgenden Schritte ausgeführt:
   3.1 Die Zeile z wird in eine neue `inline.txt`-Datei kopiert. Somit beinhaltet die `inline.txt`-Datei nur einen Funktionsnamen mit einer zugehörigen Nummer, die einen bestimmten Funktionsaufruf im Programm repräsentiert.
   3.2 Der Benchmark wird mit der höchsten Optimierungsstufe und der Option `-fWCET-inline-functions` übersetzt. Weil durch das Skript eine `inline.txt`-Datei bereitgestellt wurde, wird an dieser Stelle das manuelle Inlining-System ausgenutzt. Function Inlining wird nur für den Funktionsaufruf durchgeführt, der in der aktuellen `inline.txt`-Datei spezifiziert ist. Dabei werden auch die Werte für die Attribute, die in Abschnitt 6.4.1 vorgestellt wurden, bestimmt.
   `wcc -O3 -fWCET-inline-functions` *benchmark* (`inline.txt`-Datei ist vorhanden)
   3.3 Anschließend wird die Klassifikation des Funktionsaufrufs bestimmt. Die nach dem Function Inlining resultierende WCET_{est} wird durch die Referenz-WCET_{est} geteilt und mit 100 multipliziert. Aus dem prozentualen Ergebnis kann die Klassifikation des Funktionsaufrufs abgeleitet werden. Bei 100% gab es nach dem Function Inlining keine Veränderung in der WCET_{est}. Ist das Ergebnis kleiner 100%, so führte das Function Inlining zu einer WCET-Verbesserung, wohingegen eine WCET-Verschlechterung vorliegt, wenn das Ergebnis größer als 100% beträgt. In dieser Arbeit wurde die Grenze bei 99% gezogen, um zu verhindern, dass eine sehr geringe WCET-Verbesserung ($< 1\%$) zu einer starken Code-Vergrößerung führt. Somit wird die Klassifikation eines Funktionsaufrufs c wie folgt ermittelt:

$$Klassifikation(c) = \begin{cases} ja, & \frac{WCET_{est}\ nach\ Inlining * 100}{Referenz\text{-}WCET_{est}}\% \leq 99\% \\ nein, & sonst \end{cases} \tag{6.8}$$

   3.4 Das Beispiel wird in Form eines *Beispielvektors* zum Lerndatensatz hinzugefügt. Die bereits klassifizierten Funktionsaufrufe im Lerndatensatz können auch als Vektoren aufgefasst werden. So entspricht das i-te Beispiel für einen Funktionsaufruf im Lerndatensatz dem Vektor $(x_{i1}, \ldots, x_{ik}, y_i)$, wobei $x_{i1}, \ldots, x_{ik}$ Werte von k Attributen darstellen, die den Funktionsaufruf näher beschreiben, und y_i die Klassifikation beinhaltet.

Der Lerndatensatz wird in einer CSV-Datei gespeichert. Das Dateiformat CSV (Comma Separated Values) dient zum Austausch von tabellarischen Daten zwischen Programmen. Eine CSV-Datei beschreibt eine einfache Textdatei, in der die Zeilen einer Tabelle durch Zeilenumbrüche und die Spalten durch ein bestimmtes Trennzeichen, wie z.B. " ;", getrennt sind.

Auf diese Weise können aus jedem einzelnen Benchmark, der N Funktionsaufrufe enthält, im Allgemeinen N Beispielvektoren extrahiert werden.

6.4.3 Programme für das maschinelle Lernen

Im Folgenden werden die eingesetzten Programme für das maschinelle Lernen vorgestellt, die mit Hilfe der erzeugten Beispielvektoren eine Klassifikationsregel konstruieren können. Im letzten Schritt wird die Klassifikationsregel implementiert und als Heuristik für die WCET-gesteuerte Optimierung Function Inlining genutzt.

6.4.4 RapidMiner

Am Lehrstuhl 8 für Künstliche Intelligenz an der TU Dortmund wurden im Rahmen der Forschungstätigkeit verschiedene Programme auf dem Gebiet des maschinellen Lernens entwickelt, wie z.B. myKLR, mySVM, SVM-light oder RapidMiner. In [LS8-Software 2008] befindet sich eine vollständige Liste der am Lehrstuhl 8 entwickelten Programme.

Im Rahmen dieser Diplomarbeit wurde das Programm RapidMiner eingesetzt [RapidMiner 2008]. RapidMiner, ursprünglich YALE (Yet Another Learning Environment) genannt, ist eine Umgebung für maschinelle Lernverfahren. Das Programm wurde in der Programmiersprache Java geschrieben und ist Open-Source-Software, die unter der GNU-Lizenz AGPL[1] steht. Eine Java-API erlaubt zudem die Benutzung des Programms aus eigenen Java-Anwendungen.

In RapidMiner werden die Projekte Experimente genannt. Mittels einer graphischen Benutzeroberfläche werden Experimente aus nahezu beliebig schachtelbaren Operatoren aufgebaut. Operatoren erwarten in der Regel eine bestimmte Eingabe und berechnen anschließend eine Ausgabe, die durch andere Operatoren als Eingabe benutzt werden kann. Somit wird in RapidMiner der Prozess der Wissensentdeckung durch einen Operatorbaum modelliert. Beispiele für wichtige Operatoren sind:

- Operatoren für Ein- und Ausgabe

  Diese Operatoren sind zuständig für den Umgang mit externen Dateien. Es gibt z.B. Operatoren, die CSV-Dateien auslesen können. Das Dateiformat CSV wird in dieser Arbeit benutzt, um die gewonnenen Lerndatensätze in das Programm zu laden.

[1] GNU Affero General Public License

- Operatoren für Datenvorverarbeitung

  Mit Hilfe dieser Operatoren können z.B. Daten herausgefiltert oder durch ein bestimmtes Verfahren gewichtet werden. Zudem gibt es auch Operatoren zur Transformation von numerischen Attributen in nominale Attribute.

- Operatoren für maschinelles Lernen

  Mittels dieser Operatoren kann aus einem Lerndatensatz eine Klassifikationsregel abgeleitet werden. Es gibt Operatoren für das überwachte (Naive Bayes-Methode, Entscheidungsbäume, Zufallswälder, ...) und unüberwachte Lernen (KMeans, Agglomeratives Clustering, ...). In dieser Arbeit wurde das überwachte Lernverfahren Zufallswälder eingesetzt.

- Operatoren zur Visualisierung

  Diese Operatoren ermöglichen eine Visualisierung der Daten und Klassifikationsregeln.

- ...

Neben einer graphischen Benutzeroberfläche gibt es in RapidMiner auch ein Kommandozeilentool. Zudem bietet RapidMiner Erweiterungsmöglichkeiten durch Plug-ins.

6.4.5 GNU R

GNU R ist sowohl eine objekt-orientierte und interpretierte Programmiersprache als auch eine Programmierumgebung für statistische Datenbearbeitung und graphische Darstellungen [R Project 2008]. GNU R kommt überwiegend in den Bereichen der angewandten Statistik zum Einsatz und steht unter der GNU-Lizenz GPL[2].

GNU R wird hauptsächlich im Kommandozeilenmodus verwendet. Es gibt aber auch graphische Benutzeroberflächen. Mittels Pakete lässt sich der Funktionsumfang von GNU R erweitern. Es gibt Pakete für Klassifikationsverfahren wie z.B. Diskriminanzanalyse, Entscheidungsbäume und Zufallswälder. Das Paket für das Klassifikationsverfahren Zufallswälder heißt `randomForest`. In dieser Arbeit wurde das Paket eingesetzt, um eine Analyse der Wichtigkeit von einzelnen Attributen durchführen zu können. Das Programm RapidMiner unterstützt zwar das Klassifikationsverfahren Zufallswälder, kann aber für dieses Verfahren keine Wichtigkeitsanalyse von Attributen durchführen, was im Vorhinein nicht klar war. Aus diesem Grund wurde für eine Wichtigkeitsanalyse von Attributen das Programm GNU R benutzt. Die Ergebnisse der Wichtigkeitsanalyse von Attributen sind in Kapitel 8 enthalten.

6.4.6 Implementation von Klassifikationsregelen

Es wurde bereits erwähnt, dass in dieser Arbeit das Klassifikationsverfahren Zufallswälder benutzt wird, um eine WCET-gesteuerte Heuristik für die Optimierung Function Inlining zu generieren. Für diesen Zweck wird das Programm RapidMiner eingesetzt. Mit Hilfe eines CSV-Eingabeoperators

[2] GNU General Public License

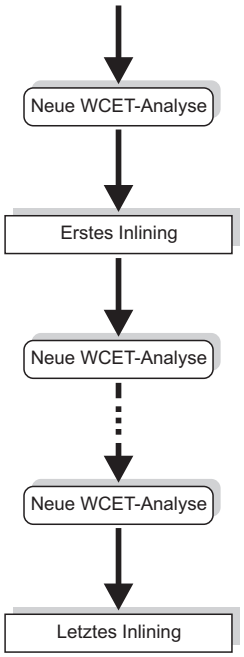


Abbildung 6.16: *Ablauf der WCET-gesteuerten Optimierung Function Inlining*

wird zuerst der in Abschnitt 6.4.2.2 aufgebaute Lerndatensatz eingelesen. Anschließend kommt das Lernverfahren Zufallswälder zum Einsatz. Die Anzahl der Entscheidungsbäume für den Zufallswald beträgt bei RapidMiner standardmäßig zehn. Weil die Klassifikationsgüte eines Zufallswaldes meist umso besser wird, je mehr einzelne Entscheidungsbäume berechnet werden, wurde die Anzahl der Entscheidungsbäume im Zufallswald auf dreizehn erhöht. Warum die Anzahl der Entscheidungsbäume nicht stark erhöht werden konnte, wird weiter unten erläutert

Die Attributauswahl in einem Entscheidungsbaum erfolgt nach dem vorgestellten Information Gain-Kriterium. Die Entscheidungsbäume im Zufallswald klassifizieren einen Funktionsaufruf, indem sie durch Abfragen der Attributwerte den vollständigen Pfad von der Wurzel bis zum Blattknoten ermitteln. Jeder Pfad von der Wurzel bis zum Blattknoten entspricht dabei einer `if-then`-Anweisung: Die Bedingung der `if-then`-Anweisung repräsentiert die einzelnen Abfragen und im then-Block wird die Klassifikation durchgeführt Daher können Entscheidungsbäume in eine Programmiersprache wie z.B. C leicht übersetzt werden. Allerdings nimmt das Implementieren eines Entscheidungsbaumes relativ viel Zeit in Anspruch. Bei Zufallswäldern ist der Zeit-Aufwand natürlich proportional zu der Anzahl der Entscheidungsbäume im Wald. Leider wurde sowohl in RapidMiner als auch in GNU R keine Möglichkeit gefunden, die gelernten Klassifikationsregeln in eine Programmiersprache zu übersetzen. Daher konnte die Anzahl der Entscheidungsbäume im Zufallswald nicht stark erhöht werden.

Jeder Entscheidungsbaum im Zufallswald wurde implementiert und wird im Quellcode durch eine Boolesche Funktion dargestellt. Beim Inlining wird für jeden Funktionsaufruf (Inline-Kandidaten), das Ergebnis eines jeden Entscheidungsbaumes abgefragt (Inlining – *ja* oder *nein*). Anschließend wird eine Mehrheitsentscheidung getroffen (vgl. Abschnitt 6.2.2). Die Auswertung der Entscheidungsbäume im Zufallswald ist effizient durchführbar und beeinflusst nicht die Laufzeit der Optimierung.

In Abb. 6.16 ist der Ablauf der WCET-gesteuerten Optimierung Function Inlining dargestellt. Damit die WCET-gesteuerte Optimierung Function Inlining auf WCET$_{est}$-Informationen zugreifen kann, wird am Anfang der Optimierung eine WCET-Analyse durchgeführt. Anschließend wird nach jedem Inlining einer Funktion eine neue WCET-Analyse gestartet, damit die WCET-gesteuerte Heuristik weiterhin auf aktuelle WCET$_{est}$-Informationen von IR-Basis-Blöcken, Funktionen und Übersetzungseinheiten zugreifen kann. Falls für einen Funktionsaufruf kein Inlining durchgeführt wird, so wird auch keine neue WCET-Analyse gestartet, weil die WCET$_{est}$-Informationen der letzten WCET-Analyse weiterhin gültig sind.

Im Rahmen dieser Diplomarbeit wurde auch eine WCET-gesteuerte Optimierung für das One-Call Function Inlining implementiert. Um eine WCET-gesteuerte Heuristik für das One-Call Function Inlining zu erzeugen, wurde lediglich am Anfang der generierten Heuristik für das *allgemeine Inlining* eine zusätzliche Bedingung eingebaut, die das Inlining nur für One-Call-Funktionen erlaubt. Handelt es sich um eine One-Call-Funktion, so wird die generierte Heuristik benutzt, um zu entscheiden, ob Function Inlining durchgeführt werden soll oder nicht. Ansonsten wird das Inlining nicht durchgeführt. Die Optimierung wird im WCC durch Angabe der Option `-fWCET-inline-one-call` aufgerufen.

Heuristik für Aufrufreihenfolge

In den WCET-gesteuerten Optimierungen Function Inlining und One-Call Function Inlining wird die Reihenfolge bei der Betrachtung der Funktionen für das Inlining durch die Aufruftiefe einer Funktion bestimmt. Falls eine Funktion im Kontrollflussgraphen früher aufgerufen werden kann als eine andere Funktion, so werden die Funktionsaufrufe zu dieser Funktion für das Inlining zuerst betrachtet. Dadurch soll mehr Potential für Standard-Optimierungen, wie z.B. Constant Propagation, geschaffen werden. Diese Vorgehensweise wird in Kapitel 8 anhand eines Beispiels erläutert und begründet.

Falls zwei Funktionen die gleiche Aufruftiefe besitzen, so wird als zweites Sortierkriterium die WCET$_{est}$ einer Funktion verwendet. In einem solchen Fall werden Funktionen mit einer größeren WCET$_{est}$ bevorzugt, um das Ziel der maximalen WCET-Reduktion bei minimalem Code-Zuwachs zu erreichen.

Die Ergebnisse der WCET-gesteuerten Optimierungen Function Inlining und One-Call Function Inlining werden in Kapitel 8 vorgestellt.

7 WCET-gesteuertes Loop Unrolling

Dieses Kapitel befasst sich mit der Entwicklung einer neuartigen WCET-gesteuerten Loop Unrolling-Optimierung. Die Optimierung soll entscheiden, um welchen Faktor eine Schleife abgerollt werden soll, sodass am Ende eine WCET-Verbesserung erreicht wird. Auch hierbei handelt es sich um ein Klassifikationsproblem. Die einzelnen Klassen können z.B. die verschiedenen Abroll-Faktoren repräsentieren. Daher können die Verfahren aus dem Bereich des maschinellen Lernens auch für diese Optimierung sinnvoll eingesetzt werden. Das zeigen auch verwandte Arbeiten, die im letzten Kapitel in Abschnitt 6.3 vorgestellt wurden. So kann die Vorgehensweise, die im letzten Kapitel präsentiert wurde, auch in diesem Kapitel angewandt werden. Allerdings soll an dieser Stelle darauf hingewiesen werden, dass der Einsatz von Verfahren aus dem Bereich des maschinellen Lernens kein Ziel dieser Diplomarbeit ist. Aus diesem Grund wird in diesem Kapitel die Heuristik für die WCET-gesteuerte Optimierung Loop Unrolling manuell entwickelt. Der Schwerpunkt dieses Kapitels liegt darin, unterschiedliche Verfahren der statischen Programmanalyse zu kombinieren, um ein aggressives Loop Unrolling zu entwickeln. Die statischen Verfahren, die hier zum Einsatz kommen, sind die statische Schleifenanalyse des WCCs und die statische WCET-Analyse.

In Abschnitt 7.1 wird die Optimierung Loop Unrolling näher vorgestellt. Dabei werden auch die Vor- und Nachteile dieser Optimierung aufgezeigt. Anschließend wird die Loop Unrolling-Optimierung der ICD-C präsentiert. In Abschnitt 7.2 werden verwandte Arbeiten diskutiert. Abschnitt 7.3 befasst sich mit der Entwicklung der WCET-gesteuerten Optimierung Loop Unrolling. Dabei werden zum Einen die Informationen ausgenutzt, die nach der Back-annotation-Phase in der ICD-C zur Verfügung stehen, zum Anderen wird die Schleifenanalyse des WCCs erstmals in einer Optimierung produktiv eingesetzt. Falls die Schleifenanalyse kein Ergebnis berechnen konnte, werden Flow Facts ausgenutzt, um Schleifeniterationsgrenzen zu bestimmen. Im letzten Abschnitt wird gezeigt, wie Schleifen mit variabler Iterationsanzahl optimiert werden können.

7.1 Loop Unrolling

Loop Unrolling ist eine traditionelle Compiler-Optimierung zur Verbesserung der Ausführungszeit von Programmen. Dabei werden Schleifen um einen bestimmten Abroll-Faktor u "abgerollt", damit sie effizienter ausgeführt werden können. Eine Schleife wird um dem Abroll-Faktor u abgerollt, indem der Schleifenrumpf $u - 1$ mal kopiert wird und die Laufvariable sowie die Schrittweite der Schleife entsprechend angepasst werden.

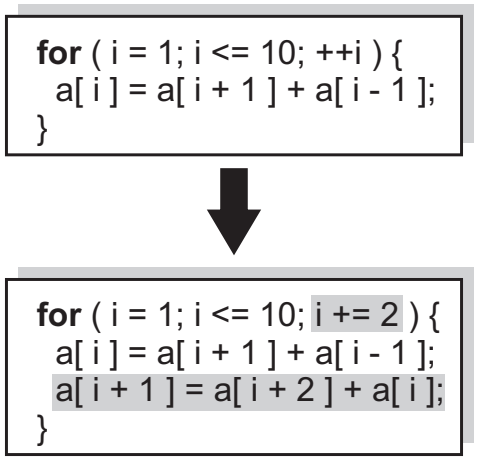


Abbildung 7.1: *Beispiel für das zweifache Abrollen einer Schleife mit konstanter Iterationsanzahl*

Abb. 7.1 zeigt ein Beispiel. Im Beispiel wird die `for`-Schleife um den Abroll-Faktor zwei abgerollt. Der Schleifenrumpf wird einmal kopiert und innerhalb der Kopie wird die Laufvariable i angepasst, indem sie um eins erhöht wird. Zum Schluss wird noch die Schrittweite der Schleife angepasst (`i += 2`). Falls die Iterationsanzahl einer Schleife nicht durch den Abroll-Faktor u ganzzahlig teilbar ist, so ist im Allgemeinen eine zusätzliche Schleife erforderlich. In Abschnitt 7.4 wird dieser Fall näher erläutert.

Durch das Abrollen einer Schleife wird die Anzahl der Iterationen der Schleife reduziert. Aus diesem Grund gibt es auch weniger Sprünge zum Schleifenkopf. Dadurch werden Pipepline Stalls durch Kontrollhazards vermieden. Infolgedessen kann das Programm in der Pipeline des Zielprozessors effizienter ausgeführt werden.

Ähnlich wie bei der Optimierung Function Inlining wird durch die Optimierung Loop Unrolling Potential für weitere Standard-Optimierungen geschaffen. Darüber hinaus wird die Parallelisierbarkeit der Instruktionen in der Schleife erhöht. Folglich können die einzelnen Ausführungseinheiten des Prozessors besser ausgenutzt werden.

Ein weiterer Vorteil der Optimierung besteht darin, dass Speicherzugriffe vermieden werden können, falls Array-Elemente in einer Iteration der abgerollten Schleife mehrfach benutzt werden. Im Beispiel aus Abb. 7.1 wird z.B. das Array-Element `a[i]` in der abgerollten Schleife zweimal benutzt. Wenn der Wert von `a[i]` nach dem ersten Speicherzugriff in einem Register gehalten wird, so kann ein Speicherzugriff vermieden werden.

Die Optmierung Loop Unrolling ist wie die Optimierung Function Inlining eine code-größen-kritische Optimierung. Folglich sind die Nachteile der Optimierung Function Inlining, die in Abschnitt 6.1 aufgezählt wurden, auch bei dieser Optimierung vorhanden. Das Abrollen einer Schleife führt zu einer Code-Vergrößerung, weil der Schleifenrumpf mindestens einmal kopiert wird (vgl. Abb. 7.1). Aufgrund der Code-Vergrößerung kann sich die Programmlaufzeit verschlechtern, weil der zusätzliche Code sowohl die Anzahl der Cache-Misses als auch den Register-Druck erhöhen kann. Ein erhöhter Register-Druck kann zu Spill-Code führen.

Die Optimierung Loop Unrolling ist somit vergleichbar mit der Optimierung Function Inlining: Beide Optimierungen können zu einer starken Verschlechterung der Programmlaufzeit führen und müssen daher mit Bedacht eingesetzt werden.

7.1.1 ICD-C Loop Unrolling

In der ICD-C-Optimierung Loop Unrolling werden nur Schleifen mit konstanter Iterationsanzahl betrachtet. Für jede Schleife wird ein möglichst großer Abroll-Faktor u statisch ermittelt, sodass

- die Iterationsanzahl der Schleife durch den Abroll-Faktor u ganzzahlig teilbar ist und
- die Anzahl der ICD-C-Ausdrücke in der abgerollten Schleife kleiner oder gleich dem Wert von `MaxUnrollingThreshold` ist.

Der Parameter `MaxUnrollingThreshold` dient dazu, den Zuwachs an Code-Größe zu beschränken. Der Wert von `MaxUnrollingThreshold` beträgt in der ICD-C standardmäßig 50, kann aber durch den Benutzer beliebig gewählt werden. Wenn kein passender Abroll-Faktor ermittelt werden konnte, so wird die Schleife nicht abgerollt. Damit der Abroll-Faktor ermittelt werden kann, müssen die Iterationsgrenzen der Schleife bekannt sein. Für diesen Zweck wird der ICD-C Loop Analyzer eingesetzt. Dieser kann allerdings nur einfache Schleifen analysieren. Aus diesem Grund wird in der neuen WCET-gesteuerten Optimierung Loop Unrolling die Schleifenanalyse des WCCs eingesetzt. Diese kann auch komplizierte Schleifen analysieren.

ICD-C Loop Unrolling wird im WCC durch Angabe der Option `-funroll-all-loops` aufgerufen. Die Optimierung wird auf der höchsten Optimierungsstufe (O3) automatisch ausgeführt.

7.2 Verwandte Arbeiten

Im Gegensatz zu den Arbeiten, die im Folgenden vorgestellt werden, beschäftigt sich diese Diplomarbeit mit der Reduktion der maximalen Laufzeit von Programmen. Weil es sich dabei um einen neuen Forschungsbereich handelt, können im Folgenden nur solche Arbeiten vorgestellt werden, die eine Verbesserung der durchschnittlichen Laufzeit von Programmen (ACET) zum Ziel haben. Ein WCET-gesteuertes Loop Unrolling für die High-Level IR wird innerhalb dieser Arbeit zum ersten Mal erforscht.

In [Koseki u.a. 1997] wird eine Heuristik für das Abrollen von verschachtelten Schleifen vorgestellt. Die Heuristik versucht für jede betrachtete Schleife mit Hilfe von Informationen, wie z.B. Abhängigkeiten zwischen Instruktionen und Wiederverwendung von Daten, einen möglichst guten Abroll-Faktor zu schätzen. Die Heuristik macht zusätzlich Gebrauch von Informationen über Hardware-Ressourcen.

In [Cohn u. Lowney 2000] werden Profil-Informationen auch durch Standard-Optimierungen wie z.B.

Loop Unrolling ausgenutzt. So werden z.B. große Schleifen, die auf dem häufig ausgeführten Pfad liegen, mit einer größeren Wahrscheinlichkeit abgerollt, wohingegen Schleifen, die nie ausgeführt werden, nicht abgerollt werden.

Ein größerer Überblick über die Optimierung Loop Unrolling ist in [Davidson u. Jinturkar 2001] enthalten. In dieser Arbeit wird betont, dass in vielen Compilern das Potential der Optimierung Loop Unrolling nicht vollständig ausgenutzt wird. Aus diesem Grund wird eine aggressive Strategie für das Abrollen von Schleifen vorgeschlagen. So werden z.B. Schleifen mit einem konstanten Abroll-Faktor von 15 abgerollt, weil festgestellt wurde, dass bei einer bestimmten RISC-Maschine mit relativ großem Befehls-Cache die meisten Sprünge bei einem Abroll-Faktor von 15 entfernt werden. Bei größeren Abroll-Faktoren ist der zusätzliche Gewinn sehr gering, weil beim Eliminieren von Sprüngen eine Sättigung erreicht wird.

Die Arbeit beschreibt zudem, wie die Optimierung Loop Unrolling implementiert werden kann und geht auch auf die Zusammenarbeit der Optimierung Loop Unrolling mit anderen Optimierungen, wie z.B. Constant Propagation oder Function Inlining, ein. Außerdem wird darauf hingewiesen, dass der Ausführungszeitpunkt der Optimierung Loop Unrolling das Ergebnis der Optimierung stark beeinflussen kann.

Auch in [Monsifrot u.a. 2002] und [Stephenson u. Amarasinghe 2005] wird die Optimierung Loop Unrolling behandelt. Die Autoren nutzen dabei Verfahren aus dem Bereich des maschinellen Lernens, um eine Heuristik für das Abrollen von Schleifen zu gewinnen (vgl. Abschnitt 6.3).

7.3 Entwicklung der WCET-gesteuerten Optimierung Loop Unrolling

Bei der Entwicklung der WCET-gesteuerten Optimierung Loop Unrolling wird inkrementell vorgegangen. Zuerst wird in Abschnitt 7.3.1 die Heuristik für die WCET-gesteuerte Optimierung Loop Unrolling vorgestellt, die den ICD-C Loop Analyzer benutzt, um die Iterationsgrenzen von Schleifen statisch zu bestimmen. Anschließend wird in Abschnitt 7.3.2 mehr Potential für die Optimierung geschaffen, indem die statische Schleifenanalyse des WCCs eingesetzt wird. Allerdings kann es vorkommen, dass die Analyse bei komplexen Programmen ohne ein Ergebnis abbricht. Aus diesem Grund wird in einem letzten Schritt in Abschnitt 7.3.3 die WCET-gesteuerte Optimierung Loop Unrolling um Flow Fact-Informationen erweitert. Diese geben u.a. Auskunft über Iterationsgrenzen von Schleifen und werden dann benutzt, wenn sowohl der ICD-C Loop Analyzer als auch die Schleifenanalyse des WCCs scheitern.

7.3.1 Entwicklung einer Heuristik

Die WCET-gesteuerte Heuristik für die Optimierung Loop Unrolling versucht für jede Schleife im Programm einen optimalen Wert für den Parameter `MaxUnrollingThreshold` zu schätzen, sodass die maximale Programmlaufzeit verringert wird. `MaxUnrollingThreshold` bestimmt die maxima-

le Anzahl der ICD-C-Ausdrücke, die in der abgerollten Schleife enthalten sein dürfen. Aus dem Wert von `MaxUnrollingThreshold` wird automatisch der Abroll-Faktor der Schleife wie folgt ermittelt: Es wird ein möglichst großer Abroll-Faktor u gewählt, sodass die Iterationsanzahl der Schleife durch den Abroll-Faktor u ganzzahlig teilbar ist und die Anzahl der ICD-C-Ausdrücke in der abgerollten Schleife kleiner oder gleich dem Wert von `MaxUnrollingThreshold` ist. Enthält z.B. eine Schleife mit konstanter Iterationsanzahl 10, insgesamt 25 ICD-C-Ausdrücke, dann wird bei einem `MaxUnrollingThreshold`-Wert von 50 der Abroll-Faktor 2 gewählt. Dieser ist durch die Iterationsanzahl der Schleife ganzzahlig teilbar und die Anzahl der ICD-C-Ausdrücke in der zweifach abgerollten Schleife beträgt genau 50 ($\leq$ `MaxUnrollingThreshold`).

Somit wird die Anzahl der ICD-C-Ausdrücke in der abgerollten Schleife durch den Parameter `MaxUnrollingThreshold` beschränkt. Je größer der Wert von `MaxUnrollingThreshold` ist, desto größer ist potentiell der statisch ermittelte Abroll-Faktor. Auf diese Weise kann ein Abroll-Faktor für Schleifen mit konstanter Iterationsanzahl bestimmt werden. Wie und um welchen Faktor Schleifen mit variabler Iterationsanzahl abgerollt werden können, wird in Abschnitt 7.4 erklärt.

Der Wert von `MaxUnrollingThreshold` wurde beim ICD-C Loop Unrolling restriktiv gewählt, um die negativen Folgen der Optimierung so weit wie möglich abzuwenden, und beträgt für jede Schleife konstant 50. Daher wird beim ICD-C Loop Unrolling jede Schleife gleich behandelt. Die WCET-geteuerte Heuristik für die Optimierung Loop Unrolling hingegen schätzt den optimalen Wert von `MaxUnrollingThreshold` für jede Schleife individuell. Die Heuristik ist in Abb. 7.2 dargestellt.

```
1  Input:  Loop
2  Output: MaxUnrollingThreshold
3
4  begin
5    MaxUnrollingThreshold := 150;
6
7    if ( is_on_Cold_Path( Loop ) )
8      MaxUnrollingThreshold := 300;
9    else if ( is_isolated( Loop ) )
10     MaxUnrollingThreshold := 200;
11
12   if ( number_of_different_DEFs( Loop ) >= 3 )
13     MaxUnrollingThreshold := 100;
14
15   if ( is_on_Infeasible_Path( Loop ) )
16     MaxUnrollingThreshold := 0;
17
18   return MaxUnrollingThreshold
19 end
```

Abbildung 7.2: *Heuristik für die WCET-gesteuerte Optimierung Loop Unrolling*

In Zeile 5 wird `MaxUnrollingThreshold` mit dem Wert 150 initialisiert. Somit beträgt der `MaxUnrollingThreshold`-Wert im Normalfall 150. Die in der Heuristik enthaltenen Werte wurden hauptsächlich durch empirische Untersuchungen bestimmt. Anhand der Abb. 7.3 soll erklärt werden, warum der `MaxUnrollingThreshold`-Wert standardmäßig 150 beträgt.

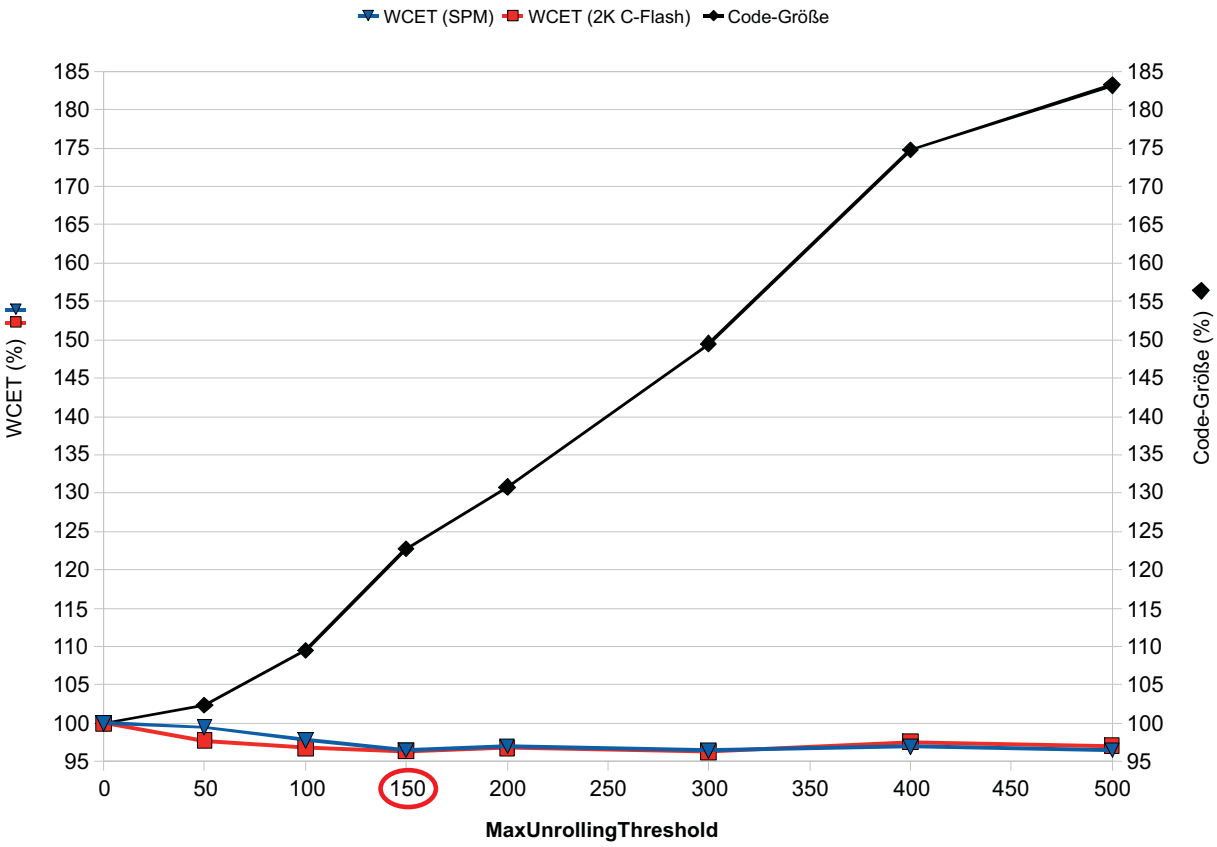


Abbildung 7.3: *Parameterwahl:* `MaxUnrollingThreshold`

Abb. 7.3 zeigt drei Liniendiagramme. Die drei Liniendiagramme zeigen den durchschnittlichen Einfluss des Parameters `MaxUnrollingThreshold` auf die Code-Größe und auf die WCET_{est} eines Programms. Der Schnitt wurde über alle Benchmarks der Benchmark-Suiten MRTC, MediaBench und NetBench gebildet. Dabei wurde der WCC mit der Optimierungsstufe O3 aufgerufen, wobei der Parameter `MaxUnrollingThreshold` in der ICD-C-Optimierung Loop Unrolling variiert wurde. Die ICD-C-Optimierung Loop Unrolling wird auf der Optimierungsstufe O3 automatisch ausgeführt. Ein Wert von 100% bedeutet, dass sich die Code-Größe bzw. die WCET_{est} eines Programms nach der ICD-C-Optimierung Loop Unrolling nicht geändert hat. Die WCET_{est} eines jeden Programms wurde für zwei unterschiedliche Speichertypen betrachtet: Scratchpad Memory (SPM) und gecachter Flash (C-Flash), d.h. alle Zugriffe auf den Befehls-Flash passieren auf dem Weg zum Prozessor einen 2 Kilobyte Befehls-Cache. Diese beiden Speichertypen werden im nächsten Kapitel bei der Evaluation der WCET-gesteuerten Optimierungen benutzt.

In Abb. 7.3 wird ersichtlich, dass die durchschnittliche WCET_{est} ab einem `MaxUnrollingThreshold`-Wert von 150 im Wesentlichen gleich bleibt. Ein größerer Wert führt im Schnitt zu einer starken Code-Vergrößerung, jedoch zu keiner signifikanten WCET-Verbesserung. Dies gilt für beide Speichertypen. Aus diesem Grund beträgt der Wert von `MaxUnrollingThreshold` bei der WCET-gesteuerten Heuristik für die Optimierung Loop Unrolling standardmäßig 150. Abhängig von der aktuell betrachteten Schleife kann dieser Wert durch die Heuristik erhöht bzw. verringert werden. So wird z.B. in Zeile 8 der Wert von `MaxUnrollingThreshold` auf 300 erhöht, falls die Schleife auf dem Cold Path liegt.

Optimierungen auf dem Cold Path haben direkte Auswirkungen auf die maximale Laufzeit eines Programms, weil der WCEP an dieser Stelle nicht wechseln kann. Eine lokale Verbesserung, die durch das Abrollen einer Schleife auf dem Cold Path entsteht, entspricht somit einer globalen Verbesserung. Daher wird auf dem Cold Path eine aggressive Strategie verfolgt, um auch größere Gewinne mit der WCET-gesteuerten Heuristik für die Optimierung Loop Unrolling zu erzielen.

Falls die Schleife nicht auf dem Cold Path liegt, so wird in Zeile 9 überprüft, ob die Schleife isoliert ist. Eine Schleife ist isoliert, wenn sie innerhalb einer Funktion nicht in einer Verzweigung enthalten ist. Für solche Schleifen wird der `MaxUnrollingThreshold`-Wert auf 200 gesetzt. Dadurch soll mehr Potential für Funktionen geschaffen werden, die in Wahrheit auf dem Cold Path liegen und aufgrund von Sicherheitsannahmen bei der Cold Path-Berechnung nicht berücksichtigt werden konnten.

Der Code in den Zeilen 12-13 dient zum Vorbeugen von Spill-Code. Falls die Schleife midestens drei unterschiedliche Variablen beinhaltet, deren Werte durch andere Werte überschrieben werden (DEF), so wird der `MaxUnrollingThreshold`-Wert zur Sicherheit auf 100 reduziert. Neue Variablen-Definitionen können den Register-Druck erhöhen, weil sich auch die Anzahl der Schreibzugriffe in der daraus resultierenden LLIR erhöht. Somit besteht die Gefahr, dass Spill-Code entsteht, der sich insbesondere innerhalb von Schleifen sehr negativ auswirken kann. Daher wird der `MaxUnrollingThreshold`-Wert auf 100 reduziert.

Schließlich werden Schleifen, die in jedem Kontext unausführbar sind, d.h. Dead Code darstellen, nicht abgerollt (Zeilen 15-16).

Zusammengefasst lässt sich sagen, dass die WCET-gesteuerte Optimierung Loop Unrolling auf der einen Seite eine aggressive Strategie für das Abrollen von Schleifen zur automatischen WCET-Reduktion verfolgt, auf der anderen Seite aber in Situationen, die potentiell zu einer WCET-Verschlechterung führen können, das Abrollen unterdrückt.

7.3.2 Statische Schleifenanalyse des WCCs

Um in der WCET-gesteuerten Optimierung Loop Unrolling den endgültigen Abroll-Faktor einer Schleife bestimmen zu können, müssen die Iterationsgrenzen der Schleife bekannt sein. Der ICD-C Loop Analyzer kann für einfache Schleifen die genaue Iterationsanzahl ermitteln (vgl. Abschnitt 3.1.1.2). Der ICD-C Loop Analyzer durchsucht das Programm nach Schleifen und versucht diese statisch auszuwerten. Die statische Schleifenanalyse der ICD-C ist jedoch sehr eingeschränkt. So werden z.B. die möglichen Werte von Variablen und Parameter, die in der Schleife enthalten sind, bei der Analyse nicht berücksichtigt. Aus diesem Grund kann der ICD-C Loop Analyzer im Allgemeinen die Iterationsgrenzen von Schleifen nur dann bestimmen, wenn die Laufvariable der Schleife durch konstante Werte modifiziert wird und die Abbruchbedingung durch konstante Werte eingeschränkt ist. Iterationsgrenzen sind jedoch für das Abrollen von Schleifen notwendig. Daher kann das Abrollen von Schleifen bei der Verwendung der statischen Schleifenanalyse der ICD-C oft nicht durchgeführt werden.

Um mehr Potential für die Optimierung Loop Unrolling zu schaffen, wird in dieser Arbeit die Schlei-

fenanalyse des WCCs eingesetzt, die von Daniel Cordes entwickelt wurde [Cordes 2008]. Dabei handelt es sich um eine statische Schleifenanalyse für ANSI-C Programme. Die Schleifenanalyse analysiert Programme auf der High-Level IR ICD-C und arbeitet auf Basis der Abstrakten Interpretation [Cousot u. Halbwachs 1978]. Die Abstrakte Interpretation beschreibt ein Verfahren der statischen Programmanalyse und bildet die theoretische Grundlage für die Schleifenanalyse. Mit Hilfe der Abstrakten Interpretation können die möglichen Werte von Variablen sicher approximiert werden. Die Schleifenanalyse des WCCs kann nicht nur die minimale und maximale Iterationsanzahl einer Schleife bestimmen, sondern auch die Iterationsanzahl für jeden Kontext einer Schleife ermitteln.

Vor der WCET-gesteuerten Optimierung Loop Unrolling wird die High-Level IR des zu analysierenden Programms an die Schleifenanalyse des WCCs übergeben. Anschließend wird die statische Schleifenanalyse durchgeführt und die Ergebnisse werden in einem Wörterbuch gespeichert. Im Anschluss wird die WCET-gesteuerte Optimierung Loop Unrolling aufgerufen. Die Optimierung kann mit Hilfe des Wörterbuchs die Iterationsgrenzen einer jeden Schleife ermitteln.

Die Schleifenanalyse des WCCs soll mehr Potential für die WCET-gesteuerte Optmierung Loop Unrolling schaffen, indem sie auch komplexe Schleifen, deren Iterationsgrenzen von Variablen abhängen, für die Optimierung zugänglich macht.

7.3.3 Ausnutzung von Flow Fact-Informationen

Die Schleifenanalyse des WCCs kann nicht alle in dieser Arbeit betrachteten Benchmarks analysieren. Insbesondere die Analyse von Programmen, in denen Pointer enthalten sind, kann dazu führen, dass die Analyse nicht terminiert. Aus diesem Grund wird durch die WCET-gesteuerte Optimierung Loop Unrolling eine maximale Analysezeit vorgegeben: Falls die Schleifenanalyse des WCCs für ein Ergebnis länger als 10 Minuten benötigt, so wird die Analyse ohne ein Ergebnis abgebrochen. In diesem Fall werden die Flow Fact-Informationen von Schleifen ausgenutzt, um notwendige Informationen für die WCET-gesteuerte Optimierung Loop Unrolling zu erhalten. Solche Informationen sind für eine WCET-Analyse zwingend notwendig. Deshalb sind normalerweise alle Schleifen in einem zu analysierenden Programm mit Flow Fact-Informationen annotiert (vgl. Abschnitt 3.1.3).

Außerdem kann es bei der Schleifenanalyse zu einer Überapproximation bei der Berechnung der Ergebnisse für Schleifeniterationsgrenzen kommen. Für das Abrollen von Schleifen sind jedoch exakte Schleifeniterationsgrenzen notwendig. Daher werden die durch die Schleifenanalyse berechneten Schleifeniterationsgrenzen mit den Flow Fact-Informationen verglichen. Falls keine Übereinstimmung vorliegt, so wird die Analyse abgebrochen.

In einem solchen Fall werden keine Flow Fact-Informationen benutzt, weil es sein kann, dass die Differenz in den Ergebnissen aufgrund einer falschen Flow Fact-Annotation durch den Benutzer zustande kommt.

7.4 Schleifen mit variabler Iterationsanzahl

Es wurde bereits erwähnt, dass sowohl die ICD-C-Optimierung Loop Unrolling als auch die WCET-gesteuerte Optimierung Loop Unrolling einen möglichst großen Abroll-Faktor wählen, sodass die Iterationsanzahl der Schleife durch den Abroll-Faktor ganzzahlig teilbar ist und die Anzahl der ICD-C-Ausdrücke in der abgerollten Schleife kleiner oder gleich dem Wert von `MaxUnrollingThreshold` ist. Wenn die Iterationsanzahl nicht durch den Abroll-Faktor ganzzahlig teilbar ist, dann kann die abgerollte Schleife nicht alle Iterationen der ursprünglichen Schleife ausführen. Das gilt insbesondere für Schleifen mit variabler Iterationsanzahl. Wird eine Schleife mit variabler Iterationsanzahl n um den Abroll-Faktor u abgerollt, so kann die abgerollte Schleife nur die ersten $\lfloor n/u \rfloor$ Iterationen der ursprünglichen Schleife ausführen. Die restlichen (maximal $u-1$) Iterationen werden in der Regel durch eine zusätzliche Schleife ausgeführt. Aufgrund der zusätzlichen Schleife entsteht zusätzlicher Overhead, der bei Schleifen mit geringer Iterationsanzahl größer sein kann als die Verbesserung, die durch das Abrollen erreicht wird. In Abb. 7.4 ist ein Beispiel für das dreifache Abrollen ($u = 3$) einer Schleife mit variabler Iterationsanzahl n zu sehen. Der Bezeichner n könnte z.B. ein Funktionsparameter sein.

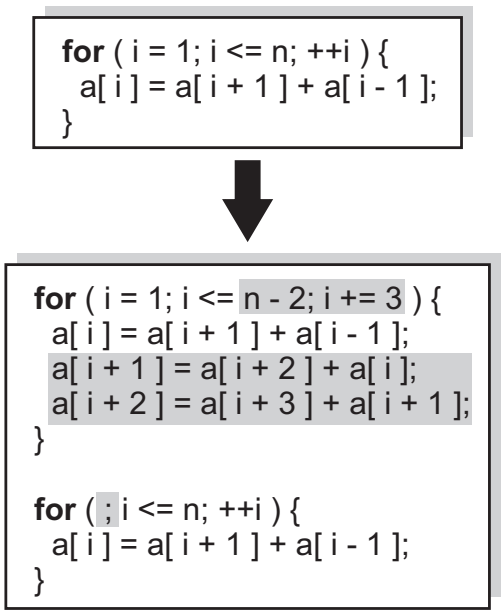


Abbildung 7.4: *Beispiel für das dreifache Abrollen einer Schleife mit variabler Iterationsanzahl*

Damit nach dem Abrollen eine WCET_{est}-Verbesserung erreicht werden kann, müssen die neuen Schleifen mit Hilfe von Flow Facts möglichst genau annotiert werden. Es gibt zwei Arten von Flow Facts: *Loopbounds* und *Flowrestrictions*. Mit Hilfe von Loopbounds kann die minimale und maximale Iterationsanzahl einer Schleife spezifiziert werden (vgl. Abschnitt 3.1.3), wohingegen Flowrestrictions die Ausführungshäufigkeit beliebiger Anweisungen gewichtet zueinander in Verhältnis setzen können. Um eine möglichst genaue Annotation durchführen zu können, müssen Flowrestrictions eingesetzt werden [Schwarzer 2007]. Da jedoch die WCET-Analyse von aiT mit Loopbounds genauer arbeiten kann als mit Flowrestrictions [Schulte 2007], müssen an dieser Stelle die ausdrucksschwächeren Loopbounds

benutzt werden. Aus diesem Grund führt das Abrollen von Schleifen mit variabler Iterationsanzahl häufig zu einer WCET_{est}-Verschlechterung. Daher wird das Erzeugen von zusätzlichen Schleifen in der WCET-gesteuerten Optimierung Loop Unrolling vermieden.

Dennoch können Schleifen mit variabler Iterationsanzahl in bestimmten Fällen abgerollt werden, ohne dass eine zusätzliche Schleife erzeugt werden muss. Besitzt z.B. eine Schleife exakt zwei Schleifeniterationsgrenzen, wobei eine Schleifeniterationsgrenze null ist, so kann die Schleife in eine neue `if`-Verzweigung eingebettet werden, dessen `if`-Bedingung der Schleifenbedingung entspricht. Außerdem kann die Schleife in eine `do-while`-Schleife transformiert werden, um die Anzahl der Sprünge zu reduzieren. So kann eine Schleife mit variabler Iterationsanzahl in eine Schleife mit konstanter Iterationsanzahl überführt werden, bei der nur die obere Iterationsgrenze betrachtet werden muss. Eine solche Schleife tritt jedoch in den betrachteten Benchmarks sehr selten auf. Daher wurde diese Optimierung aufgrund des fehlenden Potentials nicht implementiert.

Eine weitere Möglichkeit besteht darin, den Abroll-Faktor einer Schleife mit variabler Iterationsanzahl so zu wählen, dass der Abroll-Faktor dem kleinsten gemeinsamen Vielfachen (kgV) der einzelnen Iterationsgrenzen der Schleife entspricht. Falls z.B. eine Schleife 2, 3 und 6 mal iteriert, so wird als Abroll-Faktor $\text{kgV}(\{2, 3, 6\}) = 2$ gewählt. Der Vorteil besteht wiederum darin, dass keine zusätzliche Schleife erzeugt werden muss. Allerdings bieten die betrachteten Benchmarks auch für diese Optimierung wenig Potential, da der kgV bei Schleifen mit variabler Iterationsanzahl meistens eins beträgt, weil z.B. eine Iterationsgrenze der Schleife einer Primzahl oder der Zahl eins entspricht.

Eine Standard-Optimierung, die nach leichter Modifikation bei Schleifen mit variabler Iterationsanzahl eingesetzt werden kann, ist Loop Peeling [Allen u. Kennedy 2001]. Im Folgenden wird das Loop Peeling für `while`-Schleifen kurz vorgestellt. Beim Loop Peeling wird der Schleifenrumpf einer Schleife kopiert und vor die Schleife verschoben. Dadurch wird die erste Iteration einer Schleife getrennt behandelt. Anschließend wird durch eine zusätzliche `if`-Verzweigung die semantische Korrektheit der Transformation sichergestellt (vgl. Abbildung 7.5).

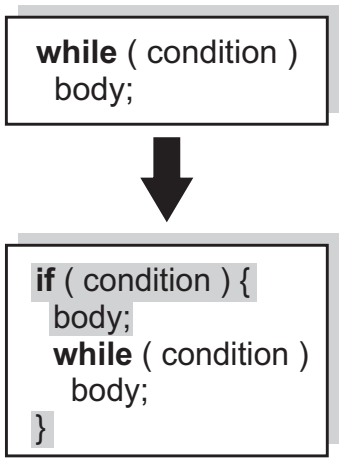


Abbildung 7.5: *Transformation einer `while`-Schleife durch Loop Peeling*

Im Folgenden wird eine Optimierung vorgestellt, das dem Loop Peeling sehr ähnlich ist: Die Idee

besteht darin, dass nicht nur der Schleifenrumpf, sondern die gesamte Schleife kopiert und vor die Schleife verschoben wird. Falls eine Schleife mit variabler Iterationsanzahl *mindestens* m mal iteriert, kann eine Kopie der Schleife mit *konstanter* Iterationsanzahl m erzeugt werden, die anschließend abgerollt werden kann.

Da die minimale Iterationsanzahl einer Schleife meistens eins beträgt und zudem eine Loop Peeling-Optimierung in der ICD-C nicht enthalten ist, wurde diese Idee nicht weiter verfolgt.

Aus den oben beschriebenen Gründen werden in dieser Arbeit nur Schleifen mit konstanter Iterationsanzahl betrachtet. Dies stellt jedoch keinen großen Nachteil dar, weil Schleifen mit variabler Iterationsanzahl in den betrachteten Benchmarks selten vorkommen. Zudem wurde erläutert, dass das Optimierungspotential bei Schleifen mit variabler Iterationsanzahl meistens gering ist.

Im nächsten Kapitel werden die Ergebnisse der WCET-gesteuerten Optimierung Loop Unrolling präsentiert.

8 Evaluation

In diesem Kapitel werden die Ergebnisse der neuen WCET-gesteuerten Optimierungen Function Inlining und Loop Unrolling vorgestellt. Für jede Optimierung wird zunächst beschrieben, wie die Ergebnisse entstehen. Anschließend werden die Ergebnisse der Optimierung für die beiden Speichertypen SPM und 2K C-Flash diskutiert.

In Abschnitt 8.1 werden zunächst die Ergebnisse der WCET-gesteuerten Optimierung Function Inlining bzw. One-Call Function Inlining vorgestellt. Anschließend werden die Ergebnisse der WCET-gesteuerten Optimierung Loop Unrolling in Abschnitt 8.2 präsentiert.

8.1 WCET-gesteuertes Function Inlining

Mit dem Skript, das in Abschnitt 6.4.2.2 vorgestellt wurde, wurde für jeden betrachteten Speichertyp (SPM und 2K C-Flash) ein Lerndatensatz erzeugt. Die Beispiele im *SPM-Lerndatensatz* wurden mit dem WCC in Kombination mit dem Speichertyp SPM gewonnen. Für die Beispiele im *C-Flash-Lerndatensatz* hingegen wurde der Speichertyp 2K C-Flash verwendet. Der Aufbau eines jeden Lerndatensatzes dauerte auf einem Intel Xeon 2,4GHz-System mit 4GB RAM ungefähr zwei Tage.

Zur Gewinnung der Beispiele wurden alle verfügbaren Benchmarks eingesetzt, weil im Allgemeinen eine größere Anzahl an Beispielen die Güte der gelernten Klassifikationsregel erhöht. Die Benchmarks stammen hauptsächlich aus den Benchmark-Suiten MRTC, MediaBench und NetBench. Einen Überblick, welcher Benchmark aus welcher Benchmark-Suite stammt und wie groß die einzelnen Benchmarks sind, gewährt Anhang A auf Seite 121. Für den SPM-Lerndatensatz konnten aus den Benchmarks nur 275 Beispiele extrahiert werden.

Der Vergleich mit den Arbeiten von [Monsifrot u.a. 2002] und [Stephenson u. Amarasinghe 2005] zeigt, dass die Anzahl der Beispiele im SPM-Lerndatensatz verhältnismäßig gering ist (vgl. Abschnitt 6.3). Der C-Flash-Lerndatensatz beinhaltet sogar noch weniger Beispiele (255), weil der Benchmark epic in Kombination mit dem Speichertyp 2K C-Flash nicht übersetzt werden konnte. Zudem gibt es in den aufgebauten Lerndatensätzen sehr wenige positive Beispiele. Ein positives bzw. negatives Beispiel beschreibt ein Beispiel für einen Funktionsaufruf c, der beim Aufbau des Lerndatensatzes

anhand der Formel 8.1 mit ja bzw. nein klassifiziert wurde (vgl. Abschnitt 6.4.2.2).

$$Klassifikation(c) = \begin{cases} ja, & \frac{WCET_{est}\ nach\ Inlining * 100}{Referenz\text{-}WCET_{est}}\% \leq 99\% \\ nein, & sonst \end{cases} \tag{8.1}$$

Somit stellen positive Beispiele Funktionsaufrufe dar, bei denen das Inlining der aufgerufenen Funktion zu einer wesentlichen Verbesserung der WCET des zu optimierenden Programms geführt hat. Tabelle 8.1 zeigt eine Übersicht der Lerndatensätze.

	SPM-Lerndatensatz	C-Flash-Lerndatensatz
Benchmarks	41	40
Beispiele	275	255
Positive Beispiele	61	35
Negative Beispiele	214	220

Tabelle 8.1: *Der SPM- und C-Flash-Lerndatensatz*

In Tabelle 8.1 fällt auf, dass die Anzahl an positiven Beispielen beim C-Flash-Lerndatensatz im Vergleich zum SPM-Lerndatensatz gering ist, weil im Schnitt die Ergebnisse für den Speichertyp 2K C-Flash schlechter waren, als die Ergebnisse für den Speichertyp SPM. Die genauen Gründe sind jedoch unbekannt. Dennoch kann angenommen werden, dass die Ursache im Speichersystem liegt.

Nachdem der Lerndatensatz aufgebaut wurde, wurde das Programm RapidMiner eingesetzt. Für jeden Speichertyp wurde mit Hilfe des Klassifikationsverfahrens Zufallswälder eine Heuristik generiert (vgl. Abschnitt 6.4.6). Dabei wurden jeweils alle Beispiele aus dem Lerndatensatz benutzt, um eine möglichst erfolgreiche Klassifikation durchführen zu können.

Somit wurden insgesamt zwei Heuristiken für die WCET-gesteuerte Optimierung Function Inlining erzeugt und implementiert: eine Heuristik für den Speichertyp SPM (im Folgenden SPM-Heuristik genannt) und eine Heuristik für den Speichertyp 2K C-Flash (im Folgenden C-Flash-Heuristik genannt). Die Heuristiken wurden auch für das WCET-gesteuerte One-Call Function Inlining benutzt (vgl. Abschnitt 6.4.6).

Abhängig vom aktuell verwendeten Speichertyp wird beim Aufruf der WCET-gesteuerten Optimierung Function Inlining (`-fWCET-inline-functions`) automatisch die passende Heuristik gewählt (SPM- oder C-Flash-Heuristik). Dies gilt auch für die WCET-gesteuerte Optimierung One-Call Function Inlining (`-fWCET-inline-one-call`).

Im Folgenden werden die Ergebnisse der WCET-gesteuerten Optimierung Function Inlining vorgestellt. Die in den Optimierungen eingesetzten Heuristiken wurden mit allen Benchmarks trainiert. Tabelle 8.2 zeigt die Ergebnisse für das WCET-gesteuerte Function Inlining.

Die Ergebnisse wurden mit der höchsten Optimierungsstufe des WCCs (O3) gewonnen. Bei *O3 ohne Inlining* wurde das ICD-C Function Inlining, das normalerweise auf der höchsten Optimierungsstufe

	SPM		2K C-Flash	
	$WCET_{est}$	Code-Größe	$WCET_{est}$	Code-Größe
O3 ohne Inlining (Referenzwert)	100,0%	100,0%	100,0%	100,0%
O3 ICD-C Inlining 50	101,7%	102,9%	100,3%	103,0%
O3 ICD-C Inlining 100	104,6%	107,6%	105,5%	107,8%
O3 ML Inlining	92,6%	94,2%	94,1%	95,3%

Tabelle 8.2: *Ergebnisse: WCET-gesteuertes Function Inlining*

automatisch ausgeführt wird, deaktiviert. Dieser Wert wird im Folgenden als Referenzwert benutzt. *O3 ICD-C Inlining 50* entspricht dem Function Inlining der ICD-C (vgl. Abschnitt 6.1.1). Beim Function Inlining der ICD-C wird das Inlining im Allgemeinen nur bei den Funktionen durchgeführt, die höchstens 50 ICD-C-Ausdrücke beinhalten. Bei *O3 ICD-C Inlining 100* wurde dieser Wert auf 100 erhöht. Da andere bekannte Compiler im Vergleich zu der ICD-C-Optimierung Function Inlining das Inlining auch bei größeren Funktionen durchführen, soll dieser Wert als zusätzlicher Vergleichswert dienen. *O3 ML Inlining* beschreibt die Heuristiken für die WCET-gesteuerte Optimierung Function Inlining, die mit Hilfe des maschinellen Lernverfahrens Zufallswälder erzeugt wurden (siehe Kapitel 6). Abhängig vom eingesetzten Speichertyp ist entweder die SPM- oder C-Flash-Heuristik gemeint.

Im Schnitt konnte die $WCET_{est}$ von Programmen gegenüber dem Function Inlining der ICD-C (*O3 ICD-C Inlining 50*) um maximal 9,1% beim Speichertyp SPM und 6,2% beim Speichertyp 2K C-Flash reduziert werden. Außerdem geht aus Tabelle 8.2 hervor, dass das WCET-gesteuerte Function Inlining im Schnitt zu einer Verringerung der Code-Größe geführt hat, weil durch das Inlining mehr Potential für Standard-Optimierungen, wie z.B. Constant Propagation, geschaffen wurde. An dieser Stelle muss erwähnt werden, dass ein Großteil der Funktionen in den betrachteten Benchmarks One-Call-Funktionen sind. Das Inlining solcher Funktionen führt zu keiner Code-Vergrößerung (vgl. Abschnitt 6.1.2).

In Abb. 8.1 auf Seite 96 und in Abb. 8.2 auf Seite 97 sind die Ergebnisse eines jeden Benchmarks für den Speichertyp SPM abgebildet. Da die Werte für den Speichertyp 2K C-Flash ähnlich sind, werden im Folgenden nur die Werte für den Speichertyp SPM betrachtet. Die 100%-Marke entspricht der $WCET_{est}$ bzw. der Code-Größe von *O3 ohne Inlining*. Es wird ersichtlich, dass die WCET-gesteuerte Optimierung Function Inlining insgesamt bessere Resultate liefert, als das Function Inlining der ICD-C. Zudem wurden Verschlechterungen in der maximalen Programmlaufzeit vermieden.

Beim Benchmark expint wird durch die WCET-gesteuerte Optimierung Function Inlining eine starke WCET-Verbesserung von 71,3% gegenüber dem Function Inlining der ICD-C erreicht. Gleichzeitig wird auch die Code-Größe um 81,8% reduziert. Dieser Benchmark besteht insgesamt aus drei Funktionen: `main`, `expint` und `foo`, wobei `expint` und `foo` One-Call-Funktionen sind. In der `main`-Funktion wird die Funktion `expint` aufgerufen. Die Funktion `expint` besteht aus mehr als 100 ICD-C-Ausdrücken. Daher kommt diese Funktion für das Function Inlining der ICD-C nicht in Frage. Die Funktion `expint` wiederum ruft die Funktion `foo` auf.

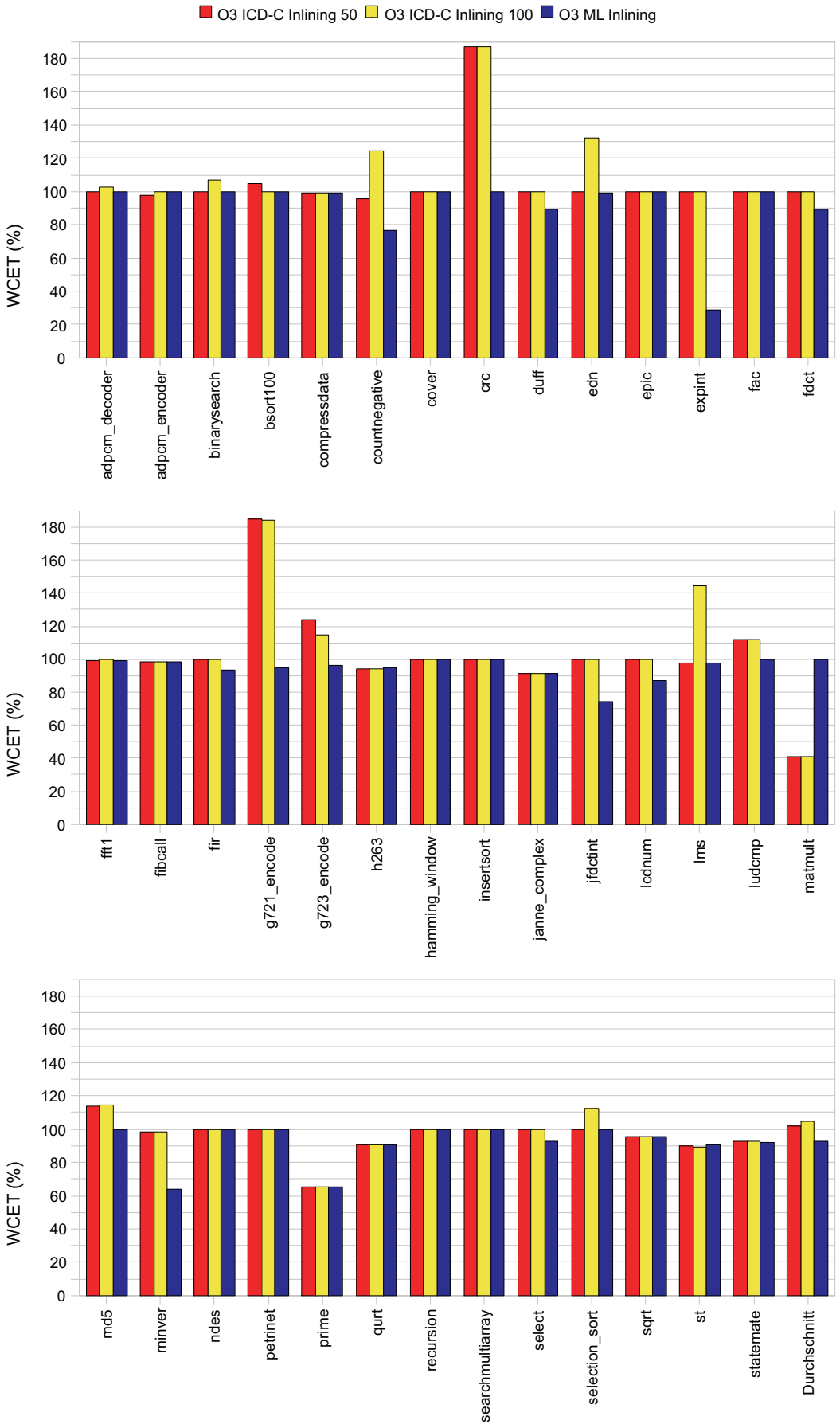


Abbildung 8.1: *WCET-gesteuertes Function Inlining (SPM) – $WCET_{est}$*

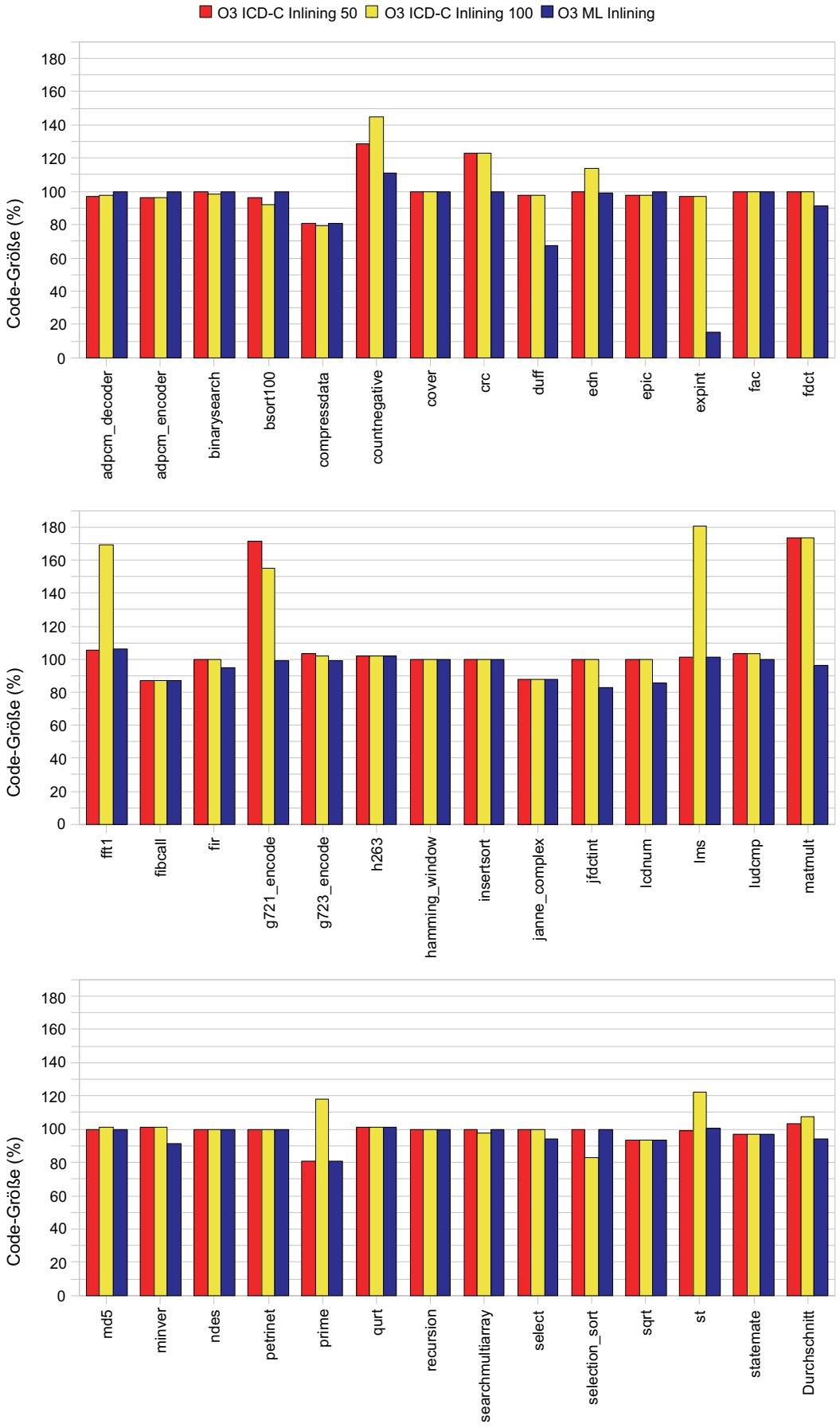


Abbildung 8.2: *WCET-gesteuertes Function Inlining (SPM) – Code-Größe*

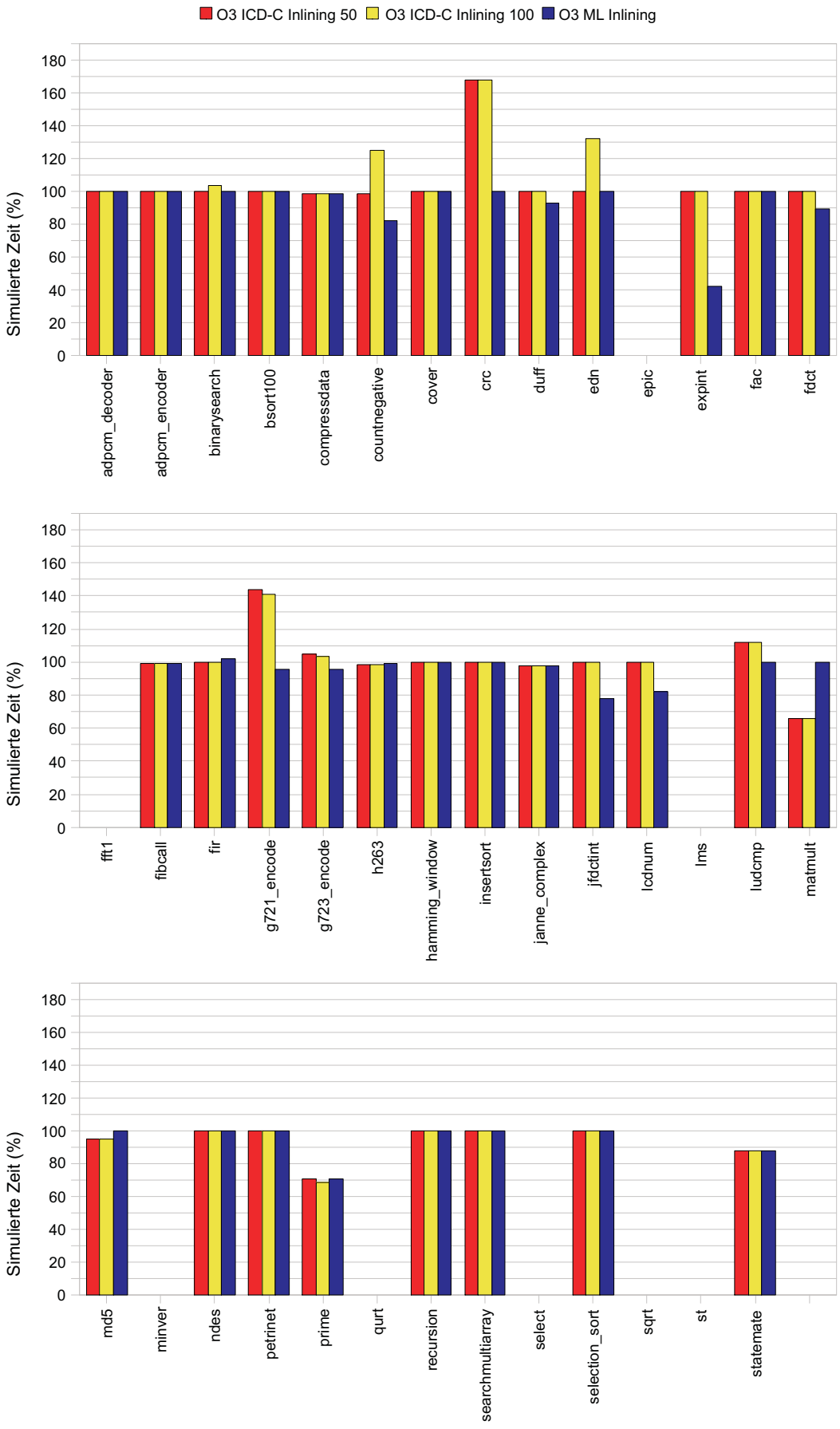


Abbildung 8.3: *WCET-gesteuertes Function Inlining (SPM) – Simulierte Zeit*

Dabei handelt es sich um eine kleine Funktion, die auch für das Function Inlining der ICD-C in Frage kommt. Somit wird beim Function Inlining der ICD-C nur für die Funktion `foo` Inlining durchgeführt. Allerdings kann der Code nach dem Inlining nicht weiter vereinfacht werden, da der einzige Parameter der Funktion `foo` von einem Parameter der Funktion `expint` abhängt. Dieser Parameter wird durch die Funktion `main` vorgegeben und ist innerhalb der Funktion `expint` unbekannt.

Bei der WCET-gesteuerten Optimierung Function Inlining wird das Inlining sowohl für die Funktion `foo` als auch für die Funktion `expint` durchgeführt. Nach dem Inlining besteht das Programm nur noch aus einer `main`-Funktion. Der vorhin unbekannte Parameter kann mit Hilfe der Optimierung Constant Propagation bis zum kopierten Funktionsrumpf der Funktion `foo` propagiert werden. Anschließend wird mit Hilfe der Optimierung Fold Constant Code zur Laufzeit des Compilers der gesamte Funktionsrumpf der Funktion `foo` statisch ausgewertet und durch eine Konstante ersetzt, die das Ergebnis repräsentiert. Aus diesem Beispiel geht hervor, dass die Heuristik nicht nur Verschlechterungen unterbindet, sondern auch zusätzliches Potential für Standard-Optimierungen schafft.

Ausgehend von dieser Beobachtung wurde in der WCET-gesteuerten Optimierung Function Inlining die Reihenfolge bei der Betrachtung der Funktionen für das Inlining modifiziert: Die Optimierung iteriert üblicherweise über alle Funktionen einer IR und überprüft für jeden Funktionsaufruf zu einer bestimmten Funktion, ob Inlining durchgeführt werden kann oder nicht. Die Reihenfolge der Funktionen wurde insofern geändert, dass Funktionen, die im Kontrollflussgraphen früher aufgerufen werden können als andere Funktion, für das Inlining zuerst betrachtet werden. Daher wird beim WCET-gesteuerten Function Inlining zuerst für die Funktion `expint` und erst danach für die Funktion `foo` Inlining durchgeführt. Dadurch wird erhofft, dass das Inlining mehr Optimierungspotential für intraprozedurale Optimierungen schafft. Andernfalls kann es passieren, dass das Inlining bei Funktionen, die weiteres Optimierungspotential schaffen (vgl. Abschnitt 6.1), wie z.B. bei der Funktion `expint`, unterbunden wird, weil deren Funktionsrumpf aufgrund vorheriger Inlining-Optimierungen stark angewachsen ist, zumal aus den aufgebauten Lerndatensätzen hervorgeht, dass das Inlining von großen Funktionen nur in den seltensten Fällen zu Gewinn führt. Daher wird das Inlining in der WCET-gesteuerten Optimierung Function Inlining für große Funktionen seltener durchgeführt als für kleine Funktionen. Falls zwei Funktionen die gleiche Aufruftiefe besitzen, so wird als zweites Sortierkriterium die WCET_{est} einer Funktion benutzt: Funktionen mit einer größeren WCET_{est} werden bevorzugt, da solche Funktionen eine größere WCET-Verbesserung versprechen. Die Änderung der Reihenfolge bewirkte z.B. beim Benchmark minver eine zusätzliche WCET-Verbesserung von 7,5%.

Die WCET-gesteuerte Optimierung Function Inlining kann jedoch nicht in jedem Benchmark das gesamte Optimierungspotential ausschöpfen. In Abb. 8.1 auf Seite 96 wird z.B. ersichtlich, dass beim Benchmark matmult kein Inlining durchgeführt wird, obwohl großes Optimierungspotential besteht. Aufgrund der geringen Anzahl an (positiven) Beispielen konnte keine ausreichend gute Heuristik generiert werden, die auch in solchen Fällen Inlining durchführt. Andererseits konnte die Heuristik bei den Benchmarks `crc` und `g721_encode` das Inlining unterbinden, sodass es zu keiner starken WCET-Verschlechterung gekommen ist.

Abb. 8.2 auf Seite 97 zeigt die Ergebnisse für die Code-Größe, wobei die 100%-Marke für die Code-

Größe von *O3 ohne Inlining* steht. Das Inlining führte größtenteils nicht zu einer Vergrößerung der Code-Größe, da die betrachteten Benchmarks sehr viele One-Call-Funktionen enthalten.

Abb. 8.3 auf Seite 98 zeigt zusätzlich die Werte für die simulierte Zeit, die durch einen Instruction Set Simulator bestimmt wurden. Die 100%-Marke entspricht der simulierten Zeit von *O3 ohne Inlining*. Mit Hilfe der simulierten Zeit soll gezeigt werden, dass die WCET-gesteuerte Optimierung als Zielfunktion die *Minimierung der maximalen Programm-Laufzeit* betrachtet, indem Unterschiede zwischen der WCET_{est} und der simulierten Zeit aufgezeigt werden. Zu Unterschieden kann es kommen, da bei einer WCET-Analyse pessimistische Annahmen getroffen werden müssen. Die Werte für die simulierte Zeit konnten jedoch nicht für alle Benchmarks ermittelt werden, weil der Simulator den Datentyp `float` nicht unterstützt. Ein Vergleich der Abb. 8.1 und 8.3 lässt erkennen, dass die WCET-gesteuerte Optimierung Function Inlining beim Benchmark fir eine Verschlechterung in der simulierten Zeit bewirkt (vgl. Abb. 8.3), wohingegen in Abb. 8.1 bei diesem Benchmark eine WCET-Reduktion erreicht wird. Somit wird bei der WCET-gesteuerten Optimierung Function Inlining eine WCET-Minimierung angestrebt, wohingegen die simulierte Zeit vernachlässigt wird, da sie beim maschinellen Lernen nicht berücksichtigt wurde.

In einem letzten Experiment wurden die Heuristiken für die beiden Speichertypen ausgetauscht. Die Heuristik, die für den Speichertyp SPM gelernt wurde, wurde für den Speichertyp 2K C-Flash benutzt und umgekehrt. Dies führte dazu, dass die WCET-Verbesserung beim Speichertyp SPM von 92,6% auf 94,0% bzw. beim Speichertyp 2K C-Flash von 94,1% auf 94,6% reduziert wurde. Dies zeigt, dass die Heuristiken auf einen bestimmten Speichertyp abgestimmt sind.

Die Kompilierzeit bei der WCET-gesteuerten Optimierung Function Inlining beträgt auf einem Intel Xeon 2,4GHz-System mit 4GB RAM im Schnitt 293,3% (SPM-Heuristik) bzw. 229,2% (C-Flash-Heuristik) der Kompilierzeit, die für die Optimierungsstufe O3 benötigt wird. Die zusätzlich benötigte Zeit kommt hauptsächlich durch die WCET-Analysen zustande. Nach jedem Inlining einer Funktion muss eine neue WCET-Analyse durchgeführt werden. Der Cold Path kann nicht ausgenutzt werden, da nach dem Inlining aktuelle WCET_{est}-Werte erforderlich sind.

8.1.1 WCET-gesteuertes One-Call Function Inlining

Im Folgenden werden die Ergebnisse für das WCET-gesteuerte One-Call Function Inlining vorgestellt. Das WCET-gesteuerte One-Call Function Inlining betrachtet nur One-Call-Funktionen und wird daher als eine separate Optimierung angesehen, die das allgemeine Function Inlining der ICD-C nicht ersetzen, sondern erweitern soll. Daher wird beim Aufruf der Optimierung im Gegensatz zum allgemeinen WCET-gesteuerten Function Inlining das Function Inlining der ICD-C nicht deaktivert. Tabelle 8.3 zeigt die Ergebnisse für das WCET-gesteuerte One-Call Function Inlining.

Als Referenzwert wird diesmal *O3 mit Inlining* benutzt. Bei *O3 Pure One-Call Inlining* wird für *alle* One-Call-Funktionen Inlining durchgeführt. *O3 ML Inlining* beschreibt die Heuristiken für die WCET-gesteuerte Optimierung One-Call Function Inlining, die mit Hilfe des maschinellen Lernverfahrens

	SPM		2K C-Flash	
	$WCET_{est}$	Code-Größe	$WCET_{est}$	Code-Größe
O3 mit Inlining (Referenzwert)	100,0%	100,0%	100,0%	100,0%
O3 Pure One-Call Inlining	95,5%	94,7%	97,7%	94,7%
O3 ML Inlining	95,0%	94,5%	95,9%	95,6%

Tabelle 8.3: *Ergebnisse: WCET-gesteuertes One-Call Function Inlining*

Zufallswälder für das allgemeine WCET-gesteuerte Function Inlining erzeugt wurden und für das One-Call Function Inlining eingesetzt werden (vgl. Abschnitt 6.4.6). Abhängig vom eingesetzten Speichertyp ist entweder die SPM- oder C-Flash-Heuristik gemeint. Da das One-Call Function Inlining eine zusätzliche Optimierung darstellt, wird jedesmal auch das Function Inlining der ICD-C ausgeführt.

Abb. 8.4 auf Seite 102 und Abb. 8.5 auf Seite 103 zeigen die Ergebnisse für jeden einzelnen Benchmark. Es wird wiederum nur der Speichertyp SPM betrachtet, da die Ergebnisse für den Speichertyp 2K C-Flash ähnlich sind. Die 100%-Marke stellt die $WCET_{est}$ bzw. die Code-Größe von *O3 mit Inlining* dar.

Im Schnitt konnte eine WCET-Verbesserung von 5% beim Speichertyp SPM und 4,1% beim Speichertyp 2K C-Flash erreicht werden. Auch bei *O3 Pure One-Call Inlining* wurde eine ähnliche WCET-Verbesserung erreicht. Im Gegensatz zu der WCET-gesteuerten Optimierung One-Call Function Inlining jedoch konnte bei einigen Benchmarks, wie z.B. binarysearch, edn oder lms, eine WCET-Verschlechterung nicht vermieden werden (vgl. Abb. 8.4 auf Seite 102).

Die Kompilierzeit bei der WCET-gesteuerten Optimierung One-Call Function Inlining beträgt auf einem Intel Xeon 2,4GHz-System mit 4GB RAM im Schnitt 215,7% (SPM-Heuristik) bzw. 203,8% (C-Flash-Heuristik) der Kompilierzeit, die für die Optimierungsstufe O3 benötigt wird. Die zusätzlich benötigte Zeit kommt hauptsächlich durch die WCET-Analysen zustande, die nach jedem Inlining einer One-Call-Funktion erforderlich sind.

Zusammengefasst lässt sich sagen, dass die WCET-gesteuerten Optimierungen Function Inlining und One-Call Function Inlining eine WCET-Verbesserung erreichen, ohne dass es in einzelnen Benchmarks zu einer Verschlechterung kommt. Weil in den betrachteten Benchmarks sehr viele One-Call-Funktionen enthalten sind, führte das Inlining in den meisten Fällen zu keiner Code-Vergrößerung. Die Code-Größe konnte sogar mit Hilfe von Standard-Optimierungen reduziert werden.

8.1.2 Fehlerratenschätzung

In diesem Abschnitt soll die Frage beantwortet werden, wie sich die Heuristiken, die mit dem Klassifikationsverfahren Zufallswälder erzeugt wurden, bei neuen Beispielen bzw. Benchmarks verhalten (d.h. wie gut können neue Beispiele klassifiziert werden). Für diesen Zweck wird das Programm RapidMiner eingesetzt. Um die Generalisierungseigenschaft der gelernten Heuristiken einschätzen zu können, können mehrere Verfahren eingesetzt werden.

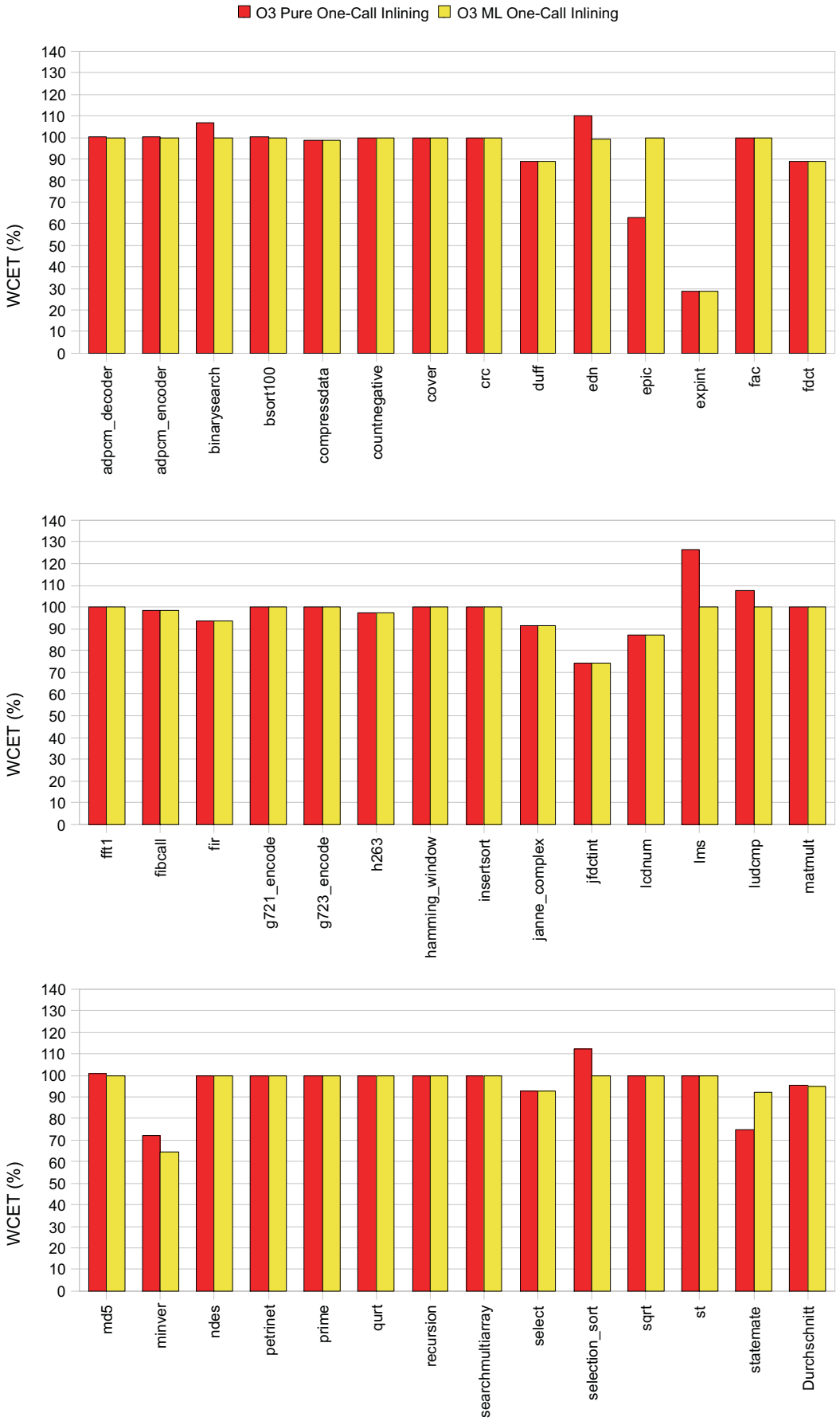


Abbildung 8.4: *WCET-gesteuertes One-Call Function Inlining (SPM) – $WCET_{est}$*

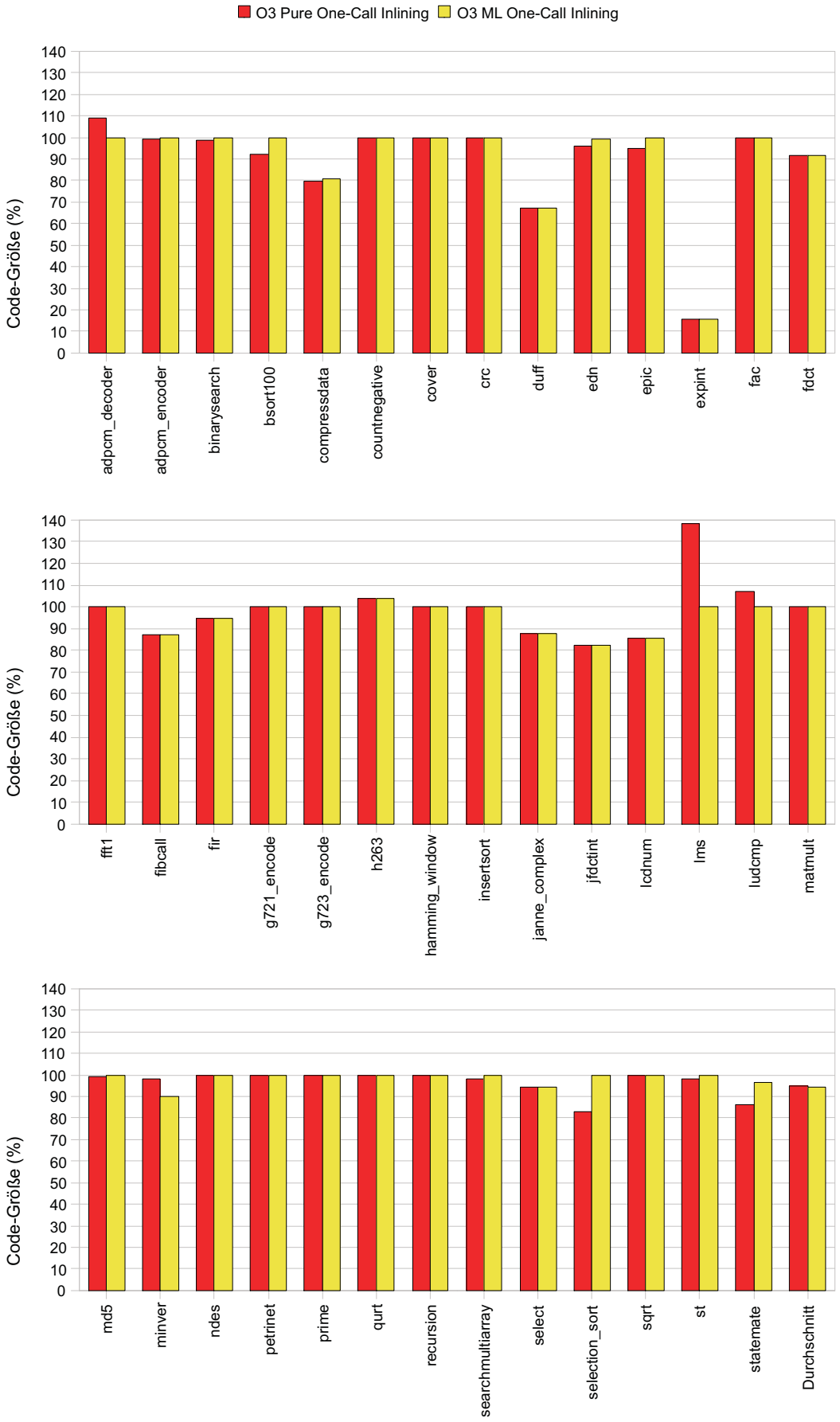


Abbildung 8.5: *WCET-gesteuertes One-Call Function Inlining (SPM) – Code-Größe*

Im Folgenden werden die Leave-One-Out Cross-Validation-Methode und die Train-and-Test-Methode vorgestellt [Morik u. Ligges SS–2007].

Leave-One-Out Cross-Validation-Methode

Ein bekanntes Testverfahren, das z.B. in [Monsifrot u.a. 2002] eingesetzt wird, ist die Leave-One-Out Cross-Validation-Methode (im Folgenden LOOCV-Methode genannt). Dabei wird ein Klassifikationsverfahren bei gegebener Größe N des Lerndatensatzes mit $N - 1$ Beispielen trainiert. Die daraus resultierende Klassifikationsregel wird mit dem Beispiel, das beim Lernen der Klassifikationsregel ausgelassen wurde, getestet. Diese Vorgehensweise wird N mal, d.h. für jedes Beispiel im Lerndatensatz, wiederholt. Anschließend wird die Genauigkeit (Fehlerrate) der Klassifikationsregel ermittelt, indem die Anzahl der Klassifikationsfehler bei den individuellen Testfällen durch N geteilt wird.

Somit kommt bei der LOOCV-Methode jedes Beispiel im Lerndatensatz als Testfall zum Einsatz. Ein weiterer großer Vorteil besteht darin, dass bei jeder Konstruktion einer Klassifikationsregel nahezu der gesamte Lerndatensatz genutzt wird. Es wird jedesmal nur ein Beispiel ausgelassen. Dadurch wird auch die Schätzung der wahren Fehlerrate genauer. Dieses Verfahren eignet sich insbesondere bei kleinen Lerndatensätzen, wie z.B. beim SPM- oder C-Flash-Lerndatensatz.

Der Nachteil der LOOCV-Methode besteht allerdings darin, dass bei einem Lerndatensatz der Größe N genau N Klassifikationsregeln erzeugt werden müssen. Daher ist diese Methode für große Lerndatensätze rechenzeitintensiv.

Die LOOCV-Methode wurde sowohl auf den SPM- als auch auf den C-Flash-Lerndatensatz angewandt. Tabelle 8.4 zeigt die Ergebnisse der LOOCV-Methode.

	SPM	2K C-Flash
Richtig klassifizierte Beispiele	83,6%	89,0%
Richtig klassifizierte positive Beispiele	34,4%	20,0%
Richtig klassifizierte negative Beispiele	97,6%	100,0%

Tabelle 8.4: *LOOCV-Methode – Genauigkeit*

Insgesamt konnten beim Speichertyp SPM 83,6% und beim Speichertyp 2K C-Flash 89,0% der Beispiele richtig klassifiziert werden. Es fällt auf, dass die Genauigkeit bei der Klassifikation positiver Beispiele im Vergleich zu der Klassifikation negativer Beispiele stark abnimmt. So wurden z.B. beim SPM-Lerndatensatz nur 34,4% der positiven Beispiele als solche erkannt. Ein möglicher Grund ist die geringe Anzahl an positiven Beispielen in den Lerndatensätzen. Insbesondere im C-Flash-Lerndatensatz gibt es sehr wenige positive Beispiele. Daher können positive Beispiele nicht so gut klassifiziert werden wie negative Beispiele. Für die WCET-gesteuerte Optimierung Function Inlining jedoch ist die richtige Klassifikation negativer Beispiele wichtiger als die richtige Klassifikation positiver Beispiele, weil das falsche Inlining zu starken Verschlechterungen in der Programmlaufzeit führen kann (vgl. Abschnitt 6.2).

Da im Rahmen der Fehlerratenschätzung auch die WCET_{est}-Werte von Benchmarks interessant sind, die nicht für das Lernen benutzt wurden, und das Implementieren von N Klassifikationsregeln bei einem Lerndatensatz der Größe N sehr viel Zeit in Anspruch nimmt, wurde in dieser Diplomarbeit zusätzlich die klassische Train-and-Test-Methode benutzt, um eine Fehlerratenschätzung durchzuführen.

Train-and-Test-Methode

Bei der Train-and-Test-Methode wird der Lerndatensatz in einen Trainings- und einen Testdatensatz aufgeteilt, z.B. in einem Verhältnis von 70-zu-30. Dieses Verhältnis ist häufig in der Literatur auffindbar. Der Trainingsdatensatz wird zur Konstruktion der Klassifikationsregel benutzt. Dabei werden die Testdaten ignoriert, um bei der Fehlerratenschätzung auf neue Daten zugreifen zu können. Anschließend wird die Klassifikationsregel auf den Testdatensatz angewandt und die Fehlerrate wird ermittelt.

Der Vorteil besteht darin, dass nur *eine* Klassifikationsregel erzeugt werden muss, um eine Fehlerratenschätzung durchzuführen. Allerdings liegt der Nachteil des Verfahrens darin, dass bei der Konstruktion der Klassifikationsregel viele Beispiele nicht benutzt werden können, weil sie im Testdatensatz enthalten sind (z.B. 30% bei einem Verhältnis von 70-zu-30). Je mehr Beispiele für die Konstruktion der Klassifikationsregel zur Verfügung stehen, desto erfolgreicher ist aber die Klassifikation neuer Daten. Daher nimmt im Vergleich zur LOOCV-Methode die Genauigkeit der Fehlerratenschätzung ab, d.h. die wahre Fehlerrate kann höher sein als die Fehlerrate, die mittels der Train-and-Test-Methode ermittelt wurde.

Dennoch wurde dieses Verfahren eingesetzt, weil das Implementieren von Klassifikationsregeln sehr aufwändig ist. Im Folgenden wird nur die Vorgehensweise für den SPM-Lerndatensatz erklärt. Beim C-Flash-Datensatz wurde analog vorgegangen.

Idealerweise sollten die Beispiele aus dem Trainings- und dem Testdatensatz unabhängig voneinander sein. Aus diesem Grund wurden *alle* Beispiele, die aus einem Benchmark stammen, entweder dem Trainings- oder dem Testdatensatz zufällig zugeordnet.

Beim Aufbau des SPM-Lerndatensatzes wurden 41 Benchmarks benutzt. Es konnten aber nur von 33 Benchmarks Beispiele gewonnen werden, weil nicht in jedem Benchmark Function Inlining durchgeführt werden kann. Aus diesen 33 Benchmarks wurden zehn Benchmarks ($\approx$ 30%) zufällig ausgewählt und alle Beispiele, die aus diesen Benchmarks gewonnen wurden, wurden zum Testdatensatz hinzugefügt. Die restlichen Beispiele wurden zum Trainingsdatensatz hinzugefügt. Daraufhin wurde das Klassifikationsverfahren Zufallswälder auf den Trainingsdatensatz angewandt. Dabei muss dasselbe Klassifikationsverfahren eingesetzt werden, das auch beim Generieren der Heurstiken für die WCET-gesteuerte Optimierung Function Inlining eingesetzt wurde (Zufallswälder). Die daraus resultierende Klassifikationsregel wurde anschließend auf den Testdatensatz angewandt, um eine Fehlerratenschätzung durchzuführen. Tabelle 8.5 zeigt die Ergebnisse.

	SPM	2K C-Flash
Richtig klassifizierte Beispiele	80,0%	84,0%
Richtig klassifizierte positive Beispiele	63,6%	54,6%
Richtig klassifizierte negative Beispiele	85,1%	92,3%

Tabelle 8.5: *Train-and-Test-Methode – Genauigkeit*

Insgesamt wurden beim Speichertyp SPM 80% und beim Speichertyp 2K C-Flash 84% der Beispiele aus dem Testdatensatz richtig klassifiziert. Es fällt wieder auf, dass die Genauigkeit bei der Klassifikation positiver Beispiele im Vergleich zu der Klassifikation negativer Beispiele abnimmt. Aus den Ergebnissen kann entnommen werden, dass das Inlining bei der WCET-gesteuerten Optimierung Function Inlining oftmals unterbunden wird, weil die generierten Heuristiken aufgrund der geringen Anzahl an positiven Beispielen in den Lerndatensätzen eine defensive Strategie verfolgen.

Im nächsten Schritt wurden die konstruierten Klassifikationsregeln implementiert und als WCET-gesteuerte Heuristik für die Optimierung Function Inlining genutzt. Anschließend wurde die Heuristik anhand der zehn Benchmarks, die nicht für das Lernen benutzt wurden, evaluiert (vgl. Abb. 8.6). Die 100%-Marke entspricht der WCET_{est} von *O3 ohne Inlining*.

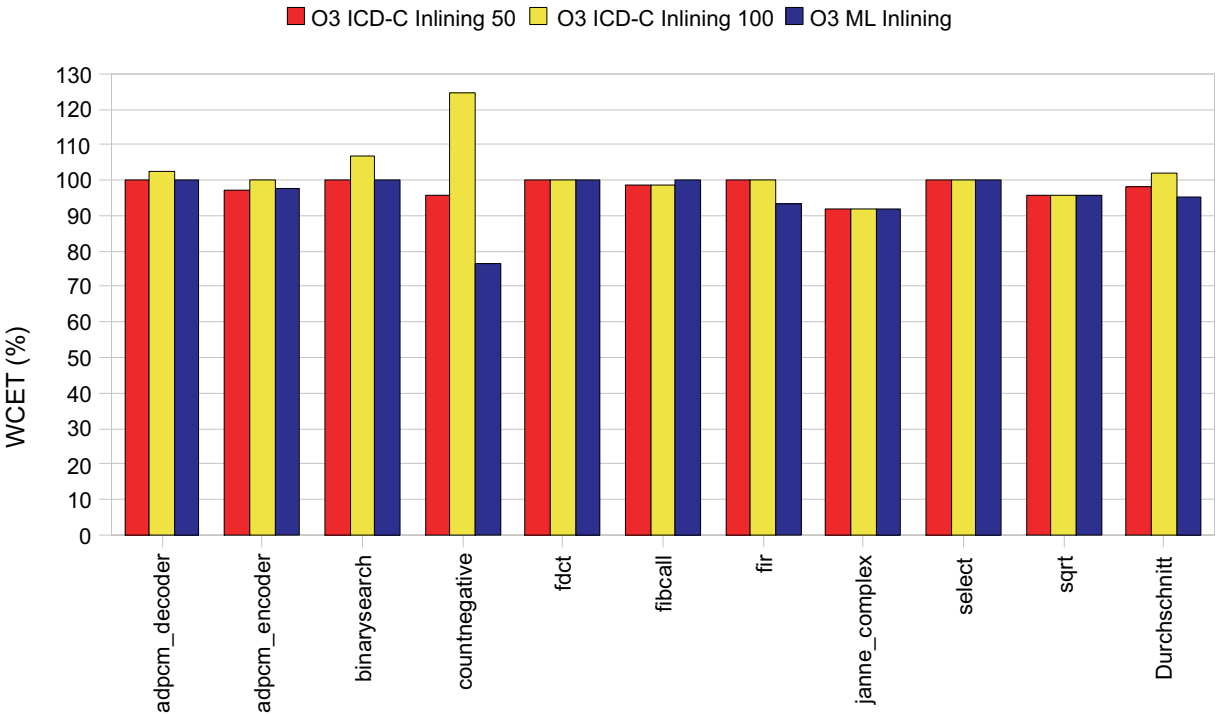


Abbildung 8.6: *WCET-gesteuertes Function Inlining (SPM) – Test*

In Tabelle 8.6 wird ersichtlich, dass beim Speichertyp SPM die Heuristik, die durch das maschinelle Lernverfahren konstruiert wurde (*O3 ML Inlining*), bei unbekannten Benchmarks im Schnitt um 2,4% besser ist als das Function Inlining der ICD-C (*O3 ICD-C Inlining 50*). Beim Speichertyp 2K C-Flash jedoch gibt es nur eine Verbesserung von 0,7% gegenüber dem Function Inlining der

ICD-C. Dennoch ist der erzielte Gewinn, insbesondere beim Speichertyp SPM, signifikant, weil die Optimierung Function Inlining oft zu einer Verschlechterung in der Programmlaufzeit führt (vgl. Abschnitt 6.2). Dies wird z.B. durch die geringe Anzahl an positiven Beispielen im SPM- und C-Flash-Lerndatensatz bekräftigt. Auch der Vergleich der Werte von *O3 ohne Inlining* und *O3 ICD-C Inlining 100* zeigt, wie stark sich die WCET_{est} aufgrund der Optimierung Function Inlining im Schnitt verschlechtern kann.

	WCET_{est} (SPM)	WCET_{est} (2K C-Flash)
O3 ohne Inlining (Referenzwert)	100,0%	100,0%
O3 ICD-C Inlining 50	97,9%	98,5%
O3 ICD-C Inlining 100	102,0%	112,3%
O3 ML Inlining	95,5%	97,8%

Tabelle 8.6: *Train-and-Test-Methode – Ergebnisse für unbekannte Benchmarks*

Auch in [Monsifrot u.a. 2002] ist der erzielte Gewinn sehr gering. Die Autoren beschäftigen sich mit dem maschinellen Lernen einer Heuristik für die High-Level-Optimierung Loop Unrolling. Obwohl die Autoren die LOOCV-Methode einsetzen und einen relativ großen Lerndatensatz besitzen, ist der erzielte Gewinn maximal 3%. Der Gewinn wird von den Autoren als wichtig angesehen, weil auch die Optimierung Loop Unrolling oft zu einer Verschlechterung in der Programmlaufzeit führt.

8.1.3 Variable Importance Measure

Das Klassifikationsverfahren Zufallswälder kann dazu benutzt werden, um eine Einschätzung der Wichtigkeit der verwendeten Attribute für die WCET-gesteuerte Optimierung Function Inlining durchzuführen (vgl. Abschnitt 6.4.1). Für diesen Zweck wurde das Programm GNU R benutzt, da das Programm RapidMiner diese Funktion für das Klassifikationsverfahren Zufallswälder nicht unterstützt (vgl. Abschnitt 6.4.3).

Die Wichtigkeit eines Attributs wird durch dessen Beitrag zur Klassentrennung bestimmt. In Abschnitt 6.2.1 wurde beschrieben, dass es mehrere Möglichkeiten gibt, die Wichtigkeit eines Attributs zu ermitteln (Kardinalitätskriteriuim, Information Gain-Kriterium). Eine weitere Möglichkeit zur Berechnung der Wichtigkeit eines Attributs ergibt sich aus dem Gini-Index. Der Gini-Index ist ein statistisches Maß zur Darstellung von Ungleichverteilungen und kann auch zur Beurteilung der Wichtigkeit von Variablen eingesetzt werden. Gemessen mit dem Gini-Index ist ein Attribut besonders wichtig, wenn es stark zur Senkung des Gini-Indexes beiträgt. Dieses Maß wird im Programm GNU R benutzt. Weitere Informationen zum Gini-Index sind z.B. in [Amiel u. Cowell 2000] zu finden.

Abb. 8.7 zeigt für den SPM-Lerndatensatz die Wichtigkeit einzelner Attribute gemessen mit dem Gini-Index. Beim C-Flash-Lerndatensatz sind die Ergebnisse sehr ähnlich. Daher werden im Folgenden nur die Ergebnisse für den SPM-Lerndatensatz betrachtet. In Abb. 8.7 steht *Call* für den Funktionsaufruf, *Caller* bezeichnet die aufrufende Funktion und *Callee* entspricht der aufgerufenen Funktion. Die Abkürzung *WCEC* bezeichnet die Anzahl der Ausführungen, z.B. eines Funktionsaufrufs, über alle

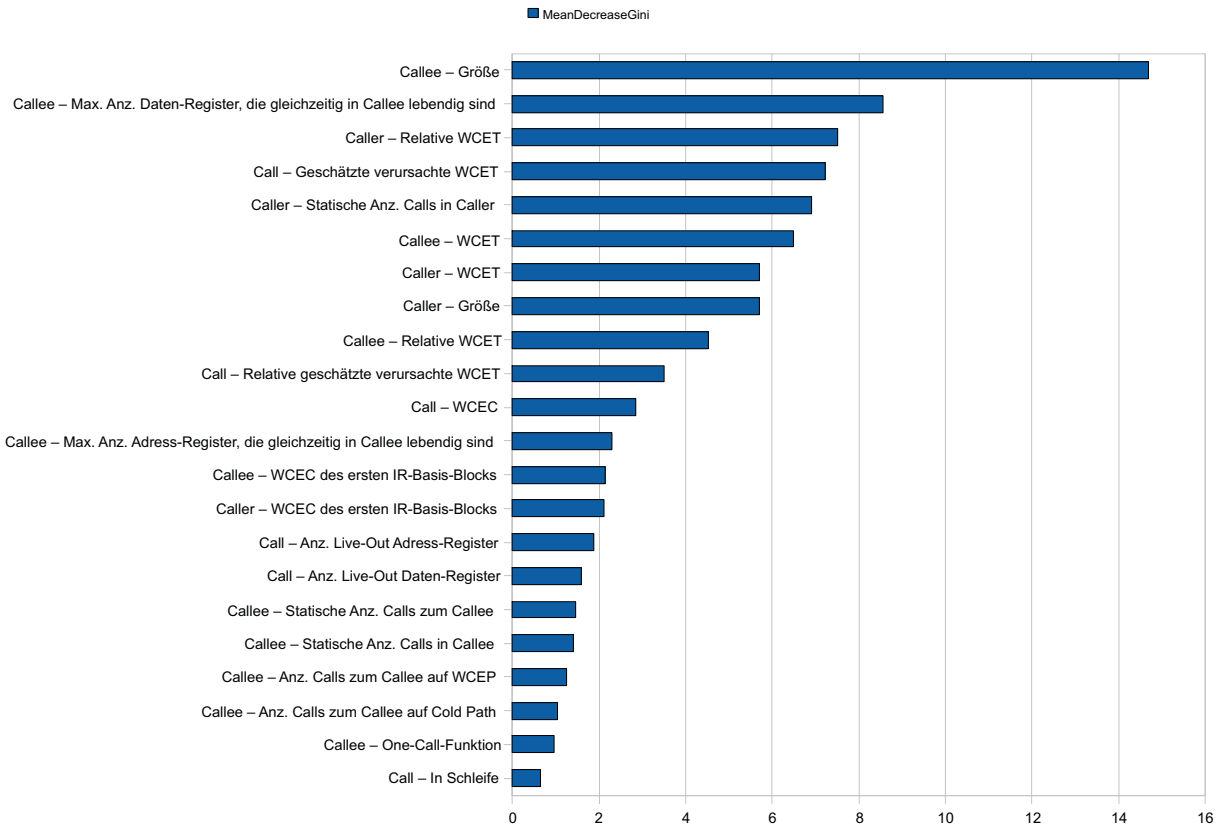


Abbildung 8.7: *Variable Importance Measure*

Kontexte hinweg (vgl. Abschnitt 4.1).

Das Attribut, das am meisten zu einer Trennung von Klassen beiträgt und zudem keine Back-annotation-Phase erfordert, ist die Größe der aufgerufenen Funktion (*Callee – Größe*). Daher wird in sehr viele Heuristiken für die Optimierung Function Inlining nur dieses eine Attribut abgefragt, wie z.B. in der ICD-C-Optimierung Function Inlining (vgl. Abschnitt 6.1.1). Es hat sich jedoch gezeigt, dass beim Inlining die alleinige Abfrage dieses Attributs oftmals zu einer starken Verschlechterung in der Programmlaufzeit führt und daher nicht ausreichend ist (vgl. Abschnitt 6.2).

Als nächstes wurde das Attribut *Callee – Maximale Anzahl an Daten-Registern, die gleichzeitig in Callee lebendig sind*, das vom Register Pressure Analyzer ermittelt wird, als besonders wichtig eingestuft. Dieses Attribut beschreibt die maximale Anzahl an Daten-Registern, die gleichzeitig in der aufgerufenen Funktion lebendig sind, und kann als Indiz für Spill-Code verwendet werden (vgl. Abschnitt 6.4.1.1).

Anschließend wurden Attribute, die Gebrauch von WCET_{est}-Informationen machen, als besonders wichtig eingestuft. Solche Informationen stehen nach der Back-annotation-Phase auf High-Level IR-Ebene zur Verfügung (siehe Kapitel 4).

Außerdem wird ersichtlich, dass nur wenige Attribute besonders wichtig sind. Die anderen Attribute scheinen aber trotzdem einen kleinen Beitrag zur Klassentrennung zu liefern. Gerade in solchen

Fällen ist der Einsatz des Klassifikationsverfahrens Zufallswälder sinnvoll, da auch Attribute, die nur einen geringen Beitrag zur Klassentrennung liefern, irgendwann auch bei der Klassifikation verwendet werden (vgl. Abschnitt 6.2.2).

8.2 WCET-gesteuertes Loop Unrolling

Bei der WCET-gesteuerten Optimierung Loop Unrolling wurden neben den Benchmark-Suiten MRTC, MediaBench und NetBench zusätzlich die Benchmark-Suiten DSPstone [Živojnović u.a. 1994] und UTDSP [Lee 1998] eingesetzt. Diese konnten beim maschinellen Lernen von Heuristiken für die WCET-gesteuerte Optimierung Function Inlining nicht berücksichtigt werden, da sie erst zu einem späteren Zeitpunkt in den WCC integriert wurden. Beim WCET-gesteuerten Loop Unrolling wurden insgesamt 96 Benchmarks betrachtet. Tabelle 8.7 zeigt die Ergebnisse der Optimierung.

	$WCET_{est}$ (SPM)	$WCET_{est}$ (2K C-Flash)	Code-Größe
O3 ohne Unrolling (Referenzwert)	100,0%	100,0%	100,0%
O3 ICD-C Unrolling	97,8%	98,1%	105,6%
O3 WCET Unrolling	95,2%	95,7%	120,5%

Tabelle 8.7: *Ergebnisse: WCET-gesteuertes Loop Unrolling*

Die Ergebnisse wurden mit der höchsten Optimierungsstufe des WCCs (O3) gewonnen. Bei *O3 ohne Unrolling* wurde das ICD-C Loop Unrolling, das normalerweise auf der höchsten Optimierungsstufe automatisch ausgeführt wird, deaktiviert. *O3 ICD-C Unrolling* entspricht dem Loop Unrolling der ICD-C (vgl. Abschnitt 7.1.1), wohingegen *O3 WCET Unrolling* der WCET-gesteuerten Optimierung Loop Unrolling entspricht, welche eine aggressive Strategie für das Abrollen von Schleifen verfolgt und zusätzlich die Schleifenanalyse des WCCs und Flow Fact-Informationen ausnutzt (siehe Kapitel 7).

Im Schnitt konnte mittels der WCET-gesteuerten Optimierung Loop Unrolling die $WCET_{est}$ der betrachteten Benchmarks von 97,8% auf 95,2% (SPM) bzw. von 98,1% auf 95,7% (2K C-Flash) reduziert werden. Somit konnte der Gewinn bzgl. der $WCET_{est}$, der durch das Abrollen von Schleifen entsteht, im Vergleich zum ICD-C Loop Unrolling mehr als verdoppelt werden.

Abb. 8.8 auf Seite 110 zeigt die Ergebnisse für jeden einzelnen Benchmark. Die 100%-Marke entspricht der $WCET_{est}$ von *O3 ohne Unrolling*. Im Folgenden werden wiederum nur die Ergebnisse für den Speichertyp SPM abgebildet, weil die Ergebnisse für den Speichertyp 2K C-Flash vergleichbar sind. Es hat sich herausgestellt, dass die Optimierung Loop Unrolling oft nicht durchgeführt werden kann. Obwohl Schleifeniterationsgrenzen bekannt sind, kann meistens kein passender Abroll-Faktor ermittelt werden, weil z.B. der `MaxUnrollingThreshold`-Wert überschritten wird. Daher konnte die WCET von Programmen insgesamt nur geringfügig verbessert werden.

Abb. 8.9 auf Seite 111 und Abb. 8.10 auf Seite 112 zeigen die Ergebnisse für 35 ausgewählte Benchmarks, die mehr Potential für das Abrollen von Schleifen bieten.

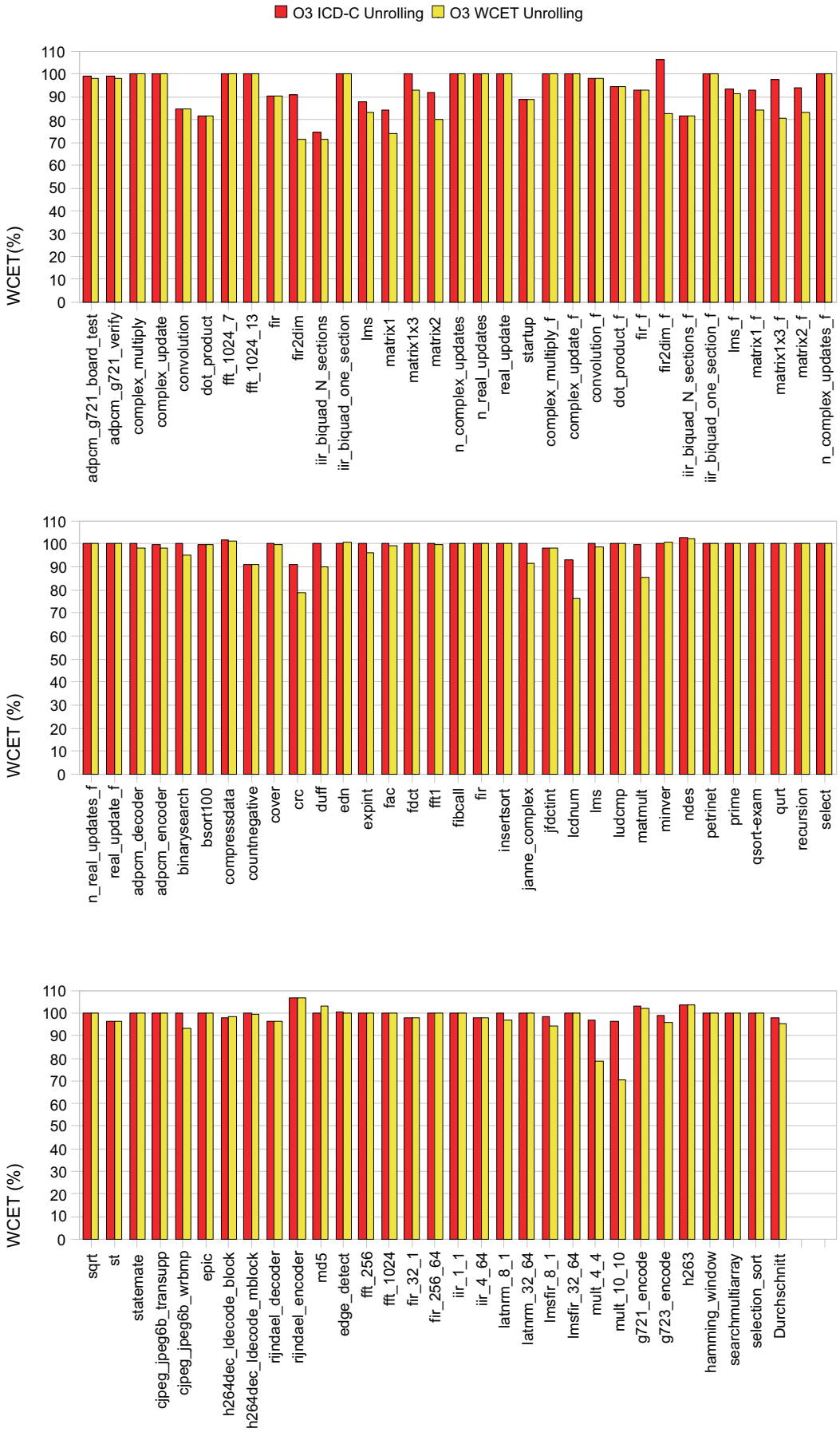


Abbildung 8.8: *WCET-gesteuertes Loop Unrolling (SPM) – $WCET_{est}$*

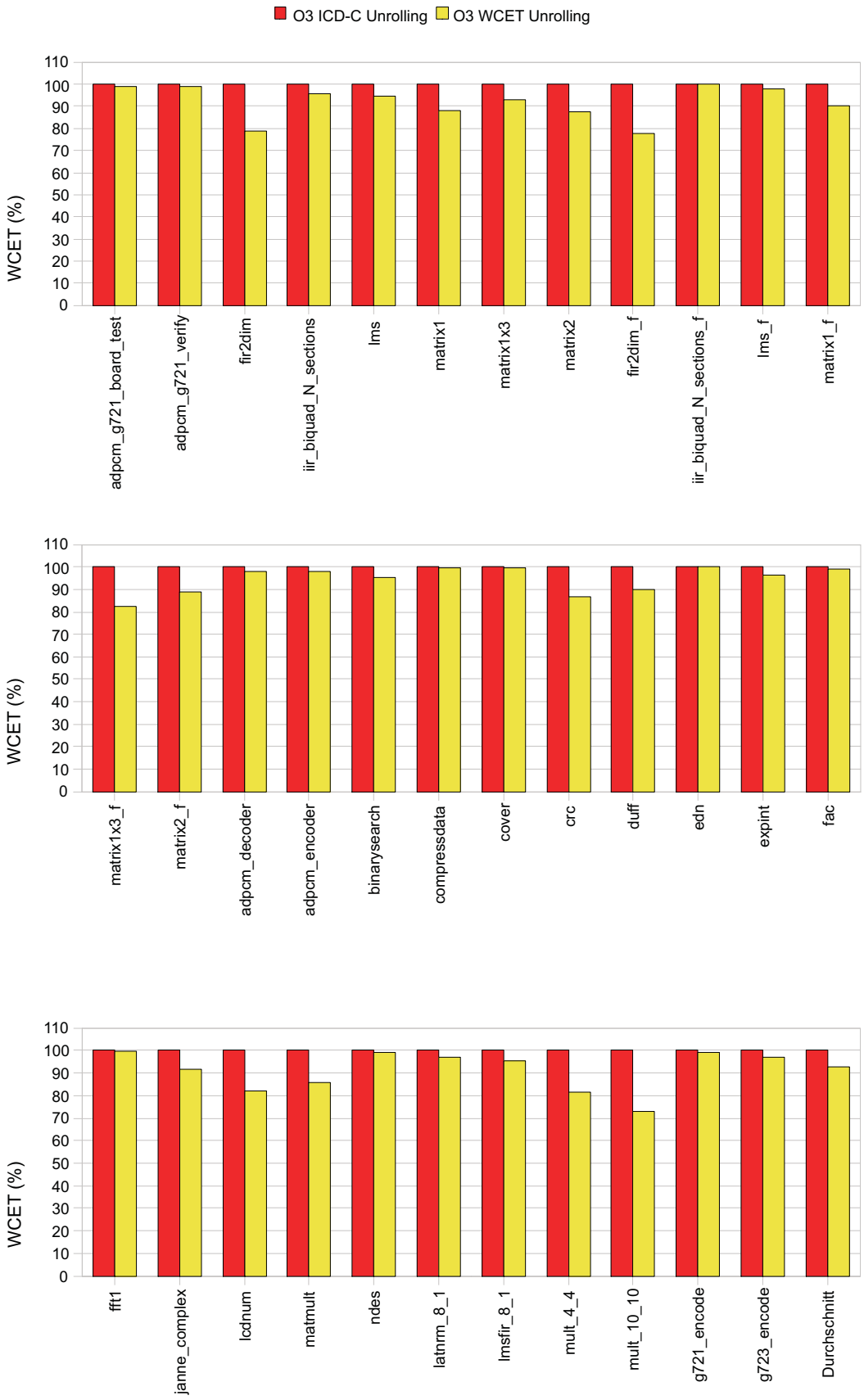


Abbildung 8.9: *WCET-gesteuertes Loop Unrolling (SPM) – $WCET_{est}$ (ausgewählte Benchmarks)*

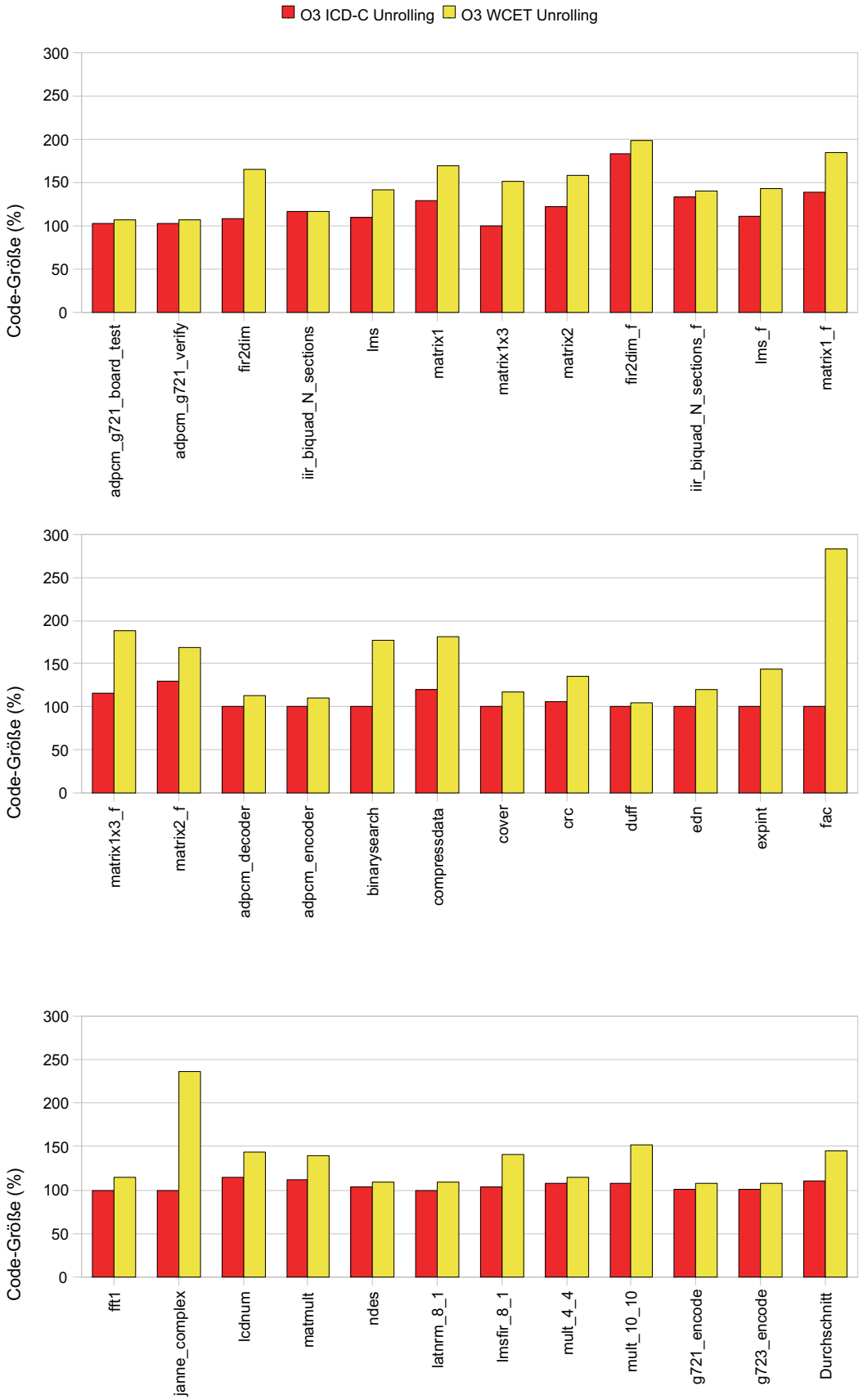


Abbildung 8.10: *WCET-gesteuertes Loop Unrolling – Code-Größe (ausgewählte Benchmarks)*

	Code-Größe
O3 ohne Unrolling (Referenzwert)	100,0%
O3 ICD-C Unrolling	110,6%
O3 WCET Unrolling	145,5%

Tabelle 8.8: *Ergebnisse: WCET-gesteuertes Loop Unrolling – Code-Größe*

In Abb. 8.9 auf Seite 111 werden die Ergebnisse der WCET-gesteuerten Optimierung Loop Unrolling mit den Ergebnissen der ICD-C-Optimierung Loop Unrolling (Referenzwert) verglichen. Im Schnitt konnte die WCET_{est} dieser Programme gegenüber dem Loop Unrolling der ICD-C um 7,5% verbessert werden.

Der Zuwachs an Code-Größe ist in Abb. 8.10 auf Seite 112 dargestellt. Die 100%-Marke entspricht der Code-Größe von *O3 ohne Unrolling*. In Kapitel 7 wurde erwähnt, dass die WCET-gesteuerte Optimierung Loop Unrolling beim Abrollen von Schleifen eine aggressive Strategie verfolgt. Das spiegelt sich auch in den Ergebnissen für die Code-Größe wieder (vgl. Tabelle 8.8). Die WCET-gesteuerte Optimierung Loop Unrolling versucht in erster Linie eine WCET-Minimierung zu erreichen. Der Zuwachs an Code-Größe ist dabei zweitrangig. Im Gegensatz zu der WCET-gesteuerten Optimierung Function Inlining wird die Code-Größe erhöht, da nach der WCET-gesteuerten Optimierung Loop Unrolling die Optimierung Remove Unused Symbols nicht angewandt werden kann.

Bei den Benchmarks fac und janne_complex wird durch die WCET-gesteuerte Optimierung Loop Unrolling die Code-Größe stark erhöht. Da es sich bei diesen Benchmarks um sehr kleine Benchmarks handelt (siehe Anhang A auf Seite 121), führt das Abrollen von Schleifen in diesen Benchmarks zu einer starken Code-Vergrößerung.

Auch bei der simulierten Zeit konnte eine Verbesserung erreicht werden (vgl. Abb. 8.11 auf Seite 114). Die 100%-Marke entspricht der simulierten Zeit von *O3 ohne Unrolling*. Weil der Simulator den Datentyp `float` nicht unterstützt, konnte die simulierte Zeit nicht für alle Benchmarks ermittelt werden. Es lässt sich erkennen, dass die WCET-gesteuerte Optimierung Loop Unrolling in erster Linie eine WCET-Reduktion anstrebt. So bewirkt die Optimierung beim Benchmark mult_10_10 eine geringe Verschlechterung in der simulierten Zeit. Gleichzeitig wird aber die WCET_{est} des Benchmarks stark reduziert (vgl. Abb. 8.8 auf Seite 110).

Zum Schluss wird in Abb. 8.12 auf Seite 115 veranschaulicht, wie groß die zusätzliche WCET-Verbesserung durch Ausnutzung der Schleifenanalyse des WCCs und Auswertung von Flow Fact-Informationen sein kann. Die 100%-Marke entspricht der WCET_{est} von *O3 ohne Unrolling*. Zudem bezeichnet *O3 WCET Unrolling ohne LA des WCCs und Flow Facts* die WCET-gesteuerte Optimierung Loop Unrolling, bei der jedoch die Schleifenanalyse des WCCs deaktiviert wurde und das Auslesen von Flow Fact-Informationen nicht unterstützt wird. Schleifeniterationsgrenzen werden nur durch den ICD-C Loop Analyzer ermittelt.

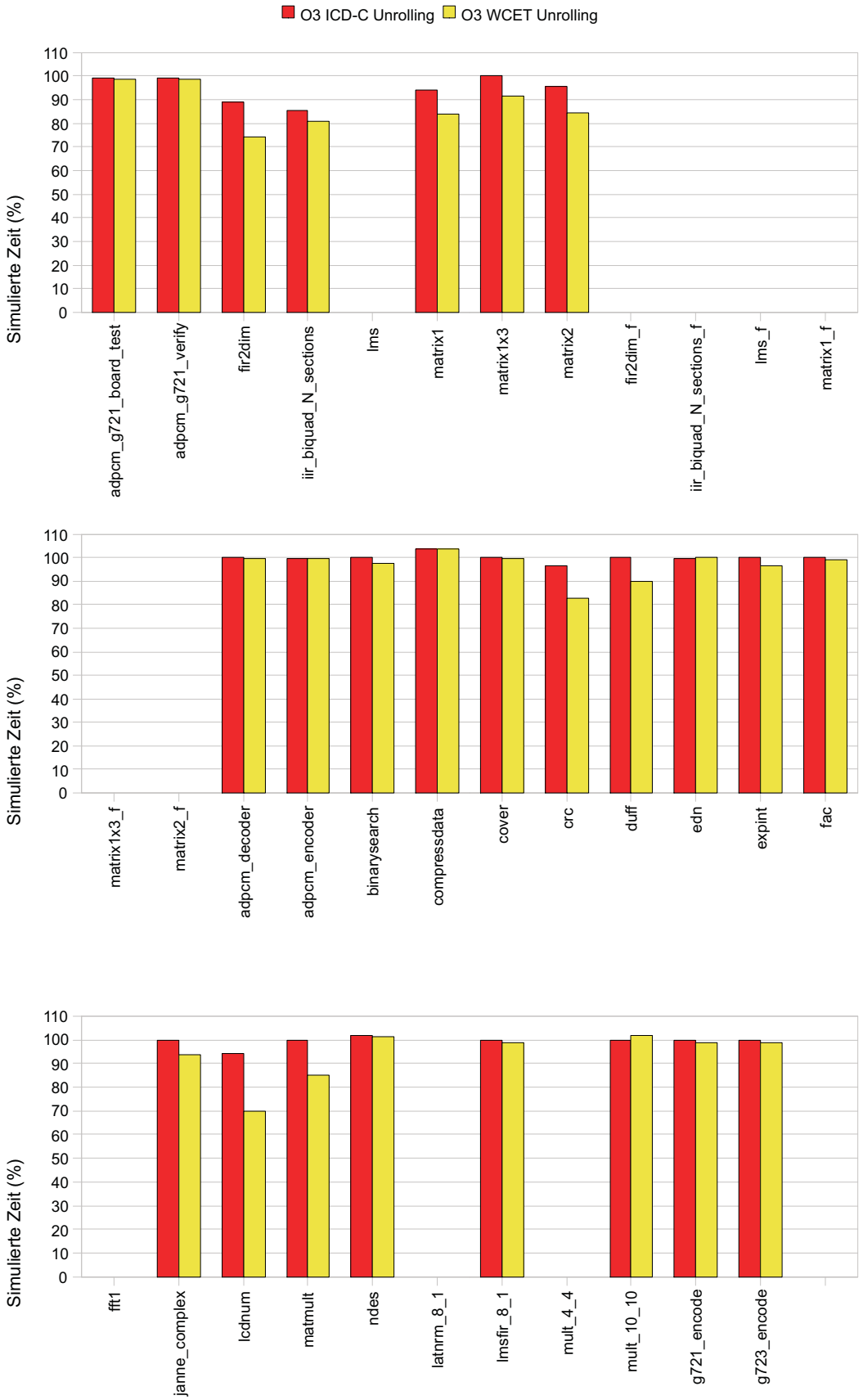


Abbildung 8.11: *WCET-gesteuertes Loop Unrolling – Simulierte Zeit (ausgewählte Benchmarks)*

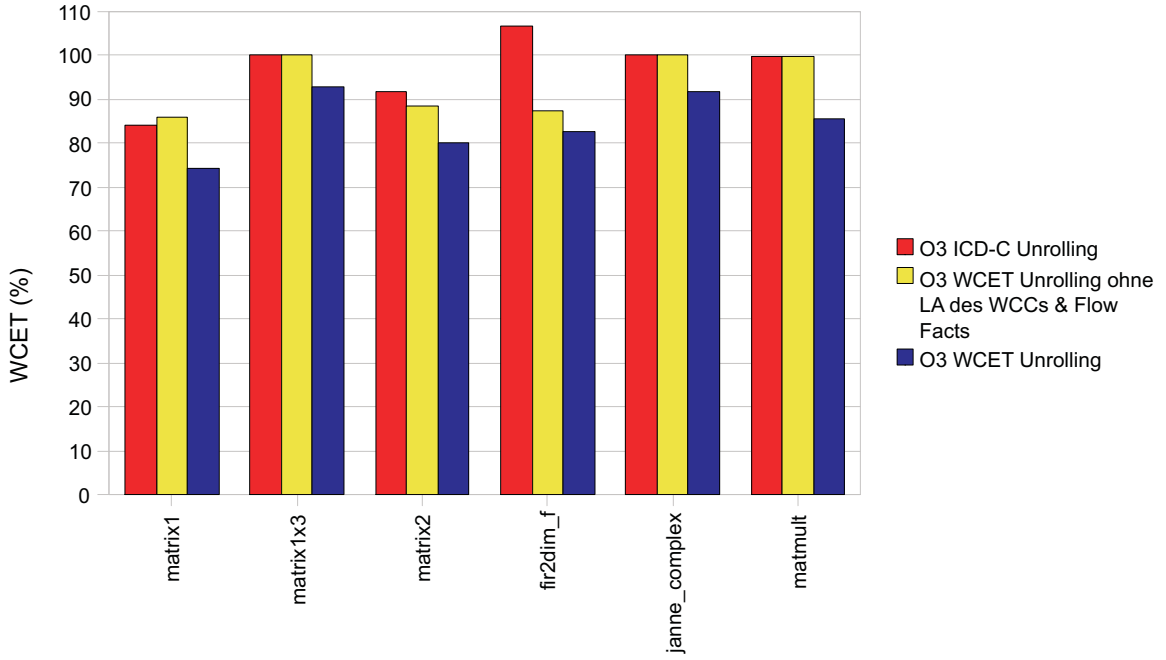


Abbildung 8.12: *Ausnutzung der Schleifenanalyse des WCCs und Auswertung von FF-Informationen*

In Abb. 8.12 sind nur solche Benchmarks abgebildet, bei denen durch Ausnutzung der Schleifenanalyse des WCCs und Auswertung von FF-Informationen zusätzlicher Gewinn erzielt werden konnte. In den meisten Benchmarks jedoch kommt der erzielt Gewinn allein durch die Heuristik für die WCET-gesteuerte Optimierung Loop Unrolling zustande, da die meisten Benchmarks einfache Schleifen beinhalten, die auch durch den ICD-C Loop Analyzer erkannt werden können.

Die Kompilierzeit bei der WCET-gesteuerten Optimierung Loop Unrolling beträgt auf einem Intel Xeon 2,4GHz-System mit 4GB RAM im Schnitt 269% der Kompilierzeit, die für die Optimierungsstufe O3 benötigt wird. Die zusätzliche Laufzeit wird hauptsächlich durch eine zusätzliche WCET-Analyse verursacht. Da die meisten Schleifen auf dem Cold Path liegen (vgl. Abschnitt 5.5), wird nur eine WCET-Analyse durchgeführt. Die Zeit, die für eine statische WCET-Analyse benötigt wird, ist meistens vernachlässigbar klein.

Zusammengefasst lässt sich sagen, dass die WCET-gesteuerte Optimierung Loop Unrolling im Vergleich zu der ICD-C-Optimierung Loop Unrolling den Gewinn bzgl. der WCET_{est} im Schnitt über alle 96 Benchmarks mehr als verdoppeln konnte. Dennoch ist der erzielte Gewinn verhältnismäßig gering. Aus den Ergebnissen geht hervor, dass die Reihenfolge der ICD-C-Optimierungen erneut durchdacht werden muss. Aufgrund von technischen Restriktionen jedoch konnte in dieser Arbeit der Ausführungzeitpunkt der WCET-gesteuerten Optimierung Loop Unrolling nicht geändert werden.

Es muss auch berücksichtigt werden, dass die Optimierung Loop Unrolling eine code-größen-kritische Optimierung ist, die, ähnlich wie bei der Optimierung Function Inlining, zu Verschlechterungen in der Programmlaufzeit führen kann. In [Monsifrot u.a. 2002] wird z.B. beschrieben, dass das Abrollen von

nur 17% (maximal 22%) der Schleifen gewinnbringend war. Daher ist das Ergebnis beachtenswert, zumal das aggressive Abrollen von Schleifen durch die WCET-gesteuerte Optimierung Loop Unrolling bei 96 Benchmarks, die aus unterschiedlichen Benchmark-Suiten stammen, zu keiner wesentlichen Verschlechterung in der Programmlaufzeit geführt hat.

9 Zusammenfassung und Ausblick

Die Transformation von detaillierten WCET_{est}-Informationen von der Low-Level IR ICD-LLIR in die High-Level IR ICD-C und die Ausnutzung dieser Informationen für die bereits bestehenden High-Level-Optimierungen Function Inlining und Loop Unrolling waren die beiden Ziele dieser Diplomarbeit.

Die wichtigsten Ergebnisse werden in Abschnitt 9.1 kurz vorgestellt. Zum Schluss wird in Abschnitt 9.2 ein Überblick über mögliche weitere Arbeiten geboten.

9.1 Ergebnisse

Der WCC ist ein WCET-optimierender C-Compiler mit integrierter WCET-Analyse. In Kapitel 2 wurden unterschiedliche Verfahren zur WCET-Analyse vorgestellt. Zudem wurde das Programm aiT näher erläutert, weil es im WCC für eine WCET-Analyse eingesetzt wird. Anschließend wurde in Kapitel 3 der WCC selbst vorgestellt.

Vor Beginn der Diplomarbeit war im WCC nur ein rudimentäres Back-annotation-Modul vorhanden, das eine Transformation von WCET_{est}-Informationen von der ICD-LLIR in die ICD-C nur für Funktionen und Übersetzungseinheiten unterstützte. Detaillierte WCET_{est}-Informationen, die in der ICD-LLIR auf Basis-Block-Ebene verfügbar sind, konnten jedoch nicht in die ICD-C transformiert werden. Daher waren solche Informationen in der ICD-C zu Beginn der Diplomarbeit nicht vorhanden. Detaillierte WCET_{est}-Informationen sind jedoch notwendig für High-Level-Optimierungen, die eine WCET-Minimierung zum Ziel haben. Aus diesem Grund wurde in Kapitel 4 das Back-annotation-Modul ausgebaut: Im ersten Schritt wurde eine Verbindung zwischen IR- und LLIR-Basis-Blöcken aufgebaut. Obgleich eine m-zu-n-Beziehung zwischen IR- und LLIR-Basis-Blöcken existiert, konnte die Verbindung erfolgreich hergestellt und verifiziert werden. Außerdem wurde in Kapitel 4 gezeigt, welche Transformationen für detaillierte WCET_{est}-Informationen notwendig sind und wie die aufgebaute Verbindung zwischen IR- und LLIR-Basis-Blöcken ausgenutzt werden kann, um detaillierte WCET_{est}-Informationen von der ICD-LLIR in die ICD-C zu transformieren. Zudem wurden, ähnlich wie bei den Flow Facts, Mechanismen zur Verfügung gestellt, die duch Low-Level-Optimierungen zur Konsistenzsicherung der Datenstrukturen des Back-annotation-Moduls genutzt werden. Neue Low-Level-Optimierungen können mit geringem Aufwand unterstützt werden.

In Kapitel 5 wurde ein *neues* Verfahren mit dem Namen Cold Path vorgestellt, das durch eine zusätzliche Auswertung der transformierten WCET_{est}-Informationen Teilpfade auf dem WCEP inden-

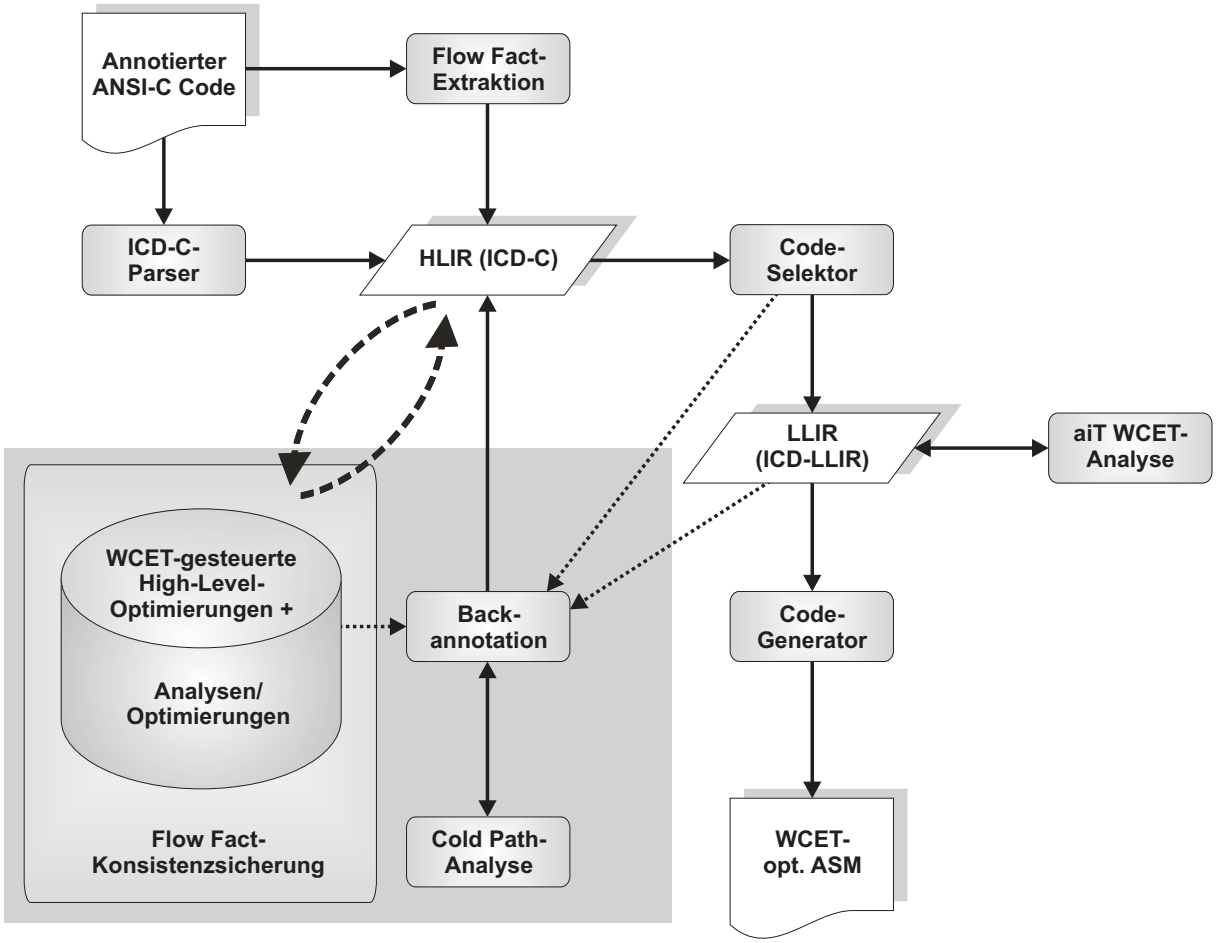


Abbildung 9.1: *WCC: Änderungen durch die Diplomarbeit*

tifiziert, die immer auf dem WCEP bleiben. Cold Path-Informationen ermöglichen die Entwicklung von effizienteren Optimierungsstrategien zur WCET-Minimierung. So können z.B. Optimierungen durch Ausnutzung von Cold Path-Informationen die Anzahl der WCET-Analysen stark reduzieren.

Mit dieser Arbeit wurde *erstmalig* ein maschinelles Lernverfahren eingesetzt, um eine Heuristik für eine High-Level-Optimierung zu generieren, die eine WCET-Minimierung zum Ziel hat:

In Kapitel 6 wurde mit Hilfe des maschinellen Lernverfahrens Zufallswälder eine WCET-gesteuerte Heuristik für die High-Level-Optimierung Function Inlining generiert. Die Heuristik macht u.a. Gebrauch von WCET_{est}-Informationen, die nach der Back-annotation-Phase zur Verfügung stehen. Ein manuelles Inlining-System wurde präsentiert und der Aufbau des Lerndatensatzes wurde automatisiert. Insgesamt wurden zwei Lerndatensätze erzeugt: ein Lerndatensatz für den Speichertyp SPM und ein Lerndatensatz für den Speichertyp 2K C-Flash. Das Programm RapidMiner wurde eingesetzt, um mittels des Klassifikationsverfahrens Zufallswälder Klassifikationsregeln automatisch zu generieren. Diese Regeln wurde anschließend implementiert und als Heuristik für die WCET-gesteuerte Optimierung Function Inlining genutzt. So konnte im Schnitt die WCET_{est} von Programmen gegenüber der Optimierung Function Inlining der ICD-C um maximal 9,1% reduziert werden. Anschließend wurde mittels der Leave-One-Out Cross-Validation-Methode und der Train-and-Test-Methode die

Generalisierungseigenschaft der gelernten Heuristik getestet. Bei der Train-and-Test-Methode wurde zusätzlich gezeigt, dass eine Heuristik, die mit dem Klassifikationsverfahren Zufallswälder erzeugt wurde, bei unbekannten Benchmarks, die nicht für das Lernen betrachtet wurden, im Schnitt eine Verbesserung von maximal 2,4% gegenüber der Optimierung Function Inlining der ICD-C erreicht.

Zudem wurde *erstmals* in dieser Arbeit die Schleifenanalyse des WCCs für eine Optimierung produktiv eingesetzt:

In Kapitel 7 wurde eine WCET-gesteuerte Optimierung Loop Unrolling inkrementell entwickelt: Im ersten Schritt wurde eine Heuristik bereitgestellt, die detaillierte WCET_{est}-Informationen und Cold Path-Informationen ausnutzt. Die vorgestellte Heuristik verfolgt eine aggressive Strategie für das Abrollen von Schleifen. Anders als bei der WCET-gesteuerten Optimierung Function Inlining wurde die Heuristik manuell erstellt. Weil die Schleifenanalyse der ICD-C nicht alle Schleifen analysieren kann, wurde im zweiten Schritt die Schleifenanalyse des WCCs in die Optimierung integriert, um mehr Potential für die WCET-gesteuerte Optimierung Loop Unrolling zu schaffen. Insbesondere bei der Analyse von komplexen Programmen, in denen Pointer enthalten sind, kann es jedoch vorkommen, dass die maximale Analysezeit überschritten wird und die Analyse ohne ein Ergebnis abgebrochen wird. Daher wurden im letzten Schritt Flow Fact-Informationen für die Optimierung zugänglich gemacht. So konnte im Schnitt der Gewinn durch das Loop Unrolling der ICD-C (2,2% im Vergleich zu O3 ohne Loop Unrolling) bei der WCET-gesteuerten Optimierung Loop Unrolling mehr als verdoppelt werden (4,8%). Werden nur solche Benchmarks betrachtet, die auch Potential für das Loop Unrolling der ICD-C als auch für das WCET-gesteuerte Loop Unrolling bieten, so beträgt die Verbesserung gegenüber der Optimierung Loop Unrolling maximal 7,5%. Es hat sich herausgestellt, dass meistens der Gewinn durch die Heuristik selbst zustande kommt, und nicht durch Ausnutzung von Informationen der Schleifenanalyse des WCCs bzw. Flow Fact-Informationen.

Abb. 9.1 zeigt den Aufbau des WCCs zum Abschluss dieser Arbeit, wobei die Änderungen aufgrund der Diplomarbeit durch eine graue Box hervorgehoben sind. Aus Gründen der Übersichtlichkeit wurde die Verbindung zwischen der LLIR (ICD-LLIR) und den Analysen/Optimierungen ausgelassen.

9.2 Ausblick

Abschließend sollen einige Aspekte genannt werden, die über die Ziele der Diplomarbeit hinausgehen und mögliche weitere Arbeiten darstellen.

Die in Kapitel 4 hergestellte Verbindung zwischen IR- und LLIR-Basis-Blöcken wird dazu genutzt, um detaillierte WCET_{est}-Informationen von der ICD-LLIR in die ICD-C zu transformieren. Die aufgebaute Verbindung zwischen IR- und LLIR-Basis-Blöcken kann auch dazu genutzt werden, um andere Informationen wie z.B. Code-Größe von der ICD-LLIR in die ICD-C zu transformieren. So könnten z.B. High-Level-Optimierungen neben detaillierten WCET_{est}-Informationen auch Informationen über die Code-Größe eines IR-Basis-Blocks ausnutzen.

In Kapitel 5 wurde der Cold Path-Algorithmus für die High-Level IR ICD-C vorgestellt. Cold Path-Informationen stehen derzeit nur in der ICD-C zur Verfügung. Im nächsten Schritt könnten Cold Path-Informationen auch für Low-Level-Optimierungen zugänglich gemacht werden. Auf diese Weise könnten bereits vorhandene Low-Level-Optimierungen, die eine WCET-Minimierung zum Ziel haben, von Cold Path-Informationen profitieren. Zudem wurden Cold Path-Informationen in dieser Arbeit nicht dazu eingesetzt, um die Anzahl der WCET-Analysen zu reduzieren. Von nun an können jedoch WCET-gesteuerte Optimierungen entwickelt werden, die durch Ausnutzung von Cold Path-Informationen die Anzahl der WCET-Analysen gezielt reduzieren. Außerdem können nun Benchmarks im Hinblick auf den Anteil des Quellcodes auf dem Cold Path evaluiert werden: So kann der Cold Path dazu genutzt werden, um die Empfindlichkeit eines Benchmarks gegenüber sich verändernde WCEPs zu beschreiben.

In Kapitel 6 wurde das maschinelle Lernverfahren Zufallswälder eingesetzt, um den Automatisierungsgrad bei der Konstruktion einer Heuristik für eine Optimierung zu erhöhen. Automatisierung ist bekanntlich ein Anliegen der Informatik. Der Prozess konnte jedoch nicht gänzlich automatisiert werden, da weder in RapidMiner noch in GNU R eine Möglichkeit gefunden wurde, die erzeugten Klassifikationsregeln automatisch in eine Programmiersprache zu übersetzen. In Zukunft könnte z.B. ein Plug-in für das Programm RapidMiner implementiert werden, das diese Aufgabe automatisiert. Somit wäre der gesamte Prozess der Konstruktion einer Heuristik automatisiert. Dies würde mehrere Vorteile mit sich bringen: Nach jeder größeren Änderung am Compiler könnte so die Heuristik automatisch angepasst werden. Die Leave-One-Out Cross-Validation-Methode, die im Vergleich zu der Train-and-Test-Methode bessere Ergebnisse liefert, könnte durchgehend, d.h. auch für die WCET-Abschätzung, eingesetzt werden. Außerdem könnte die Anzahl der Entscheidungsbäume im Zufallswald ohne großen Aufwand erhöht werden. In Abschnitt 6.2.2 wurde erwähnt, dass eine große Anzahl an Entscheidungsbäumen im Allgemeinen die Klassifikationsgüte des Zufallswaldes verbessert. Die Klassifikationsgüte kann weiter verbessert werden, indem die Anzahl der Beispiele im Lerndatensatz erhöht wird. Für diesen Zweck müssen jedoch neue Benchmarks in den WCC integriert werden. Darüber hinaus kann die Frage beantwortet werden, ob andere Verfahren aus dem Bereich des maschinellen Lernens, wie z.B. SVM oder NN, zu besseren Ergebnissen führen.

In Kapitel 7 wurde eine Heuristik für das WCET-gesteuerte Loop Unrolling manuell entwickelt. Allerdings können Verfahren aus dem Bereich des maschinellen Lernens auch für diese Optimierung sinnvoll eingesetzt werden. Des Weiteren haben die Ergebnisse gezeigt, dass die Reihenfolge der ICD-C-Optimierungen verbesserungswürdig ist, da das Abrollen von Schleifen selten zu einer wesentlichen Verbesserung der Programmlaufzeit führt.

A Realworld-Benchmarks – Code-Größe

Tabelle A.1: *Realworld-Benchmarks – Code-Größe*

Benchmark	Benchmark-Suite	Code-Größe bei O0 [Bytes]
adpcm_g721_board_test	DSPstone_fixed_point	7790
adpcm_g721_board_test	DSPstone_fixed_point	7790
adpcm_g721_verify	DSPstone_fixed_point	7822
complex_multiply	DSPstone_fixed_point	382
complex_update	DSPstone_fixed_point	286
convolution	DSPstone_fixed_point	136
dot_product	DSPstone_fixed_point	110
fft_1024_7	DSPstone_fixed_point	786
fft_1024_13	DSPstone_fixed_point	822
fir	DSPstone_fixed_point	222
fir2dim	DSPstone_fixed_point	860
iir_biquad_N_sections	DSPstone_fixed_point	330
iir_biquad_one_section	DSPstone_fixed_point	202
lms	DSPstone_fixed_point	440
matrix1	DSPstone_fixed_point	352
matrix1x3	DSPstone_fixed_point	110
matrix2	DSPstone_fixed_point	408
n_complex_updates	DSPstone_fixed_point	688
n_real_updates	DSPstone_fixed_point	514
real_update	DSPstone_fixed_point	118
startup	DSPstone_fixed_point	332
complex_multiply	DSPstone_floating_point	410
complex_update	DSPstone_floating_point	596
convolution	DSPstone_floating_point	174
dot_product	DSPstone_floating_point	260
fir	DSPstone_floating_point	258
fir2dim	DSPstone_floating_point	1030
iir_biquad_N_sections	DSPstone_floating_point	268
iir_biquad_one_section	DSPstone_floating_point	224
lms	DSPstone_floating_point	480
matrix1	DSPstone_floating_point	376
matrix1x3	DSPstone_floating_point	212
matrix2	DSPstone_floating_point	434
n_complex_updates	DSPstone_floating_point	712
n_real_updates	DSPstone_floating_point	534
real_update	DSPstone_floating_point	302
adpcm_decoder	MRTC	2820
adpcm_encoder	MRTC	2902
binarysearch	MRTC	136
bsort100	MRTC	220
compressdata	MRTC	1272
countnegative	MRTC	328
cover	MRTC	2702
crc	MRTC	902
duff	MRTC	316
edn	MRTC	4610

Benchmark	Benchmark-Suite	Code-Größe bei O0 [Bytes]
expint	MRTC	798
fac	MRTC	62
fdct	MRTC	1790
fft1	MRTC	1890
fibcall	MRTC	58
fir	MRTC	332
insertsort	MRTC	296
janne_complex	MRTC	98
jfdctint	MRTC	2042
lcdnum	MRTC	482
lms	MRTC	1364
ludcmp	MRTC	1354
matmult	MRTC	342
minver	MRTC	1452
ndes	MRTC	2166
petrinet	MRTC	6092
prime	MRTC	222
qurt	MRTC	806
recursion	MRTC	62
select	MRTC	908
sqrt	MRTC	238
st	MRTC	798
statemate	MRTC	12082
cjpeg_jpeg6b_transupp	MediaBench	7224
cjpeg_jpeg6b_wrbmp	MediaBench	682
epic	MediaBench	11752
h264dec_ldecode_block	MediaBench	16050
h264dec_ldecode_macroblock	MediaBench	18520
rijndael_decoder	MediaBench	13866
rijndael_encoder	MediaBench	13446
md5	NetBench	4896
edge_detect	UTDSP	2276
fft_256	UTDSP	2514
fft_1024	UTDSP	2512
fir_32_1	UTDSP	1394
fir_256_64	UTDSP	1486
iir_1_1	UTDSP	1442
iir_4_64	UTDSP	1874
latnrm_8_1	UTDSP	1540
latnrm_32_64	UTDSP	1520
lmsfir_8_1	UTDSP	1568
lmsfir_32_64	UTDSP	1830
mult_4_4	UTDSP	1452
mult_10_10	UTDSP	1484
g721_encode	misc	7010
g723_encode	misc	7004
h263	misc	3326
hamming_window	misc	266
searchmultiarray	misc	204
selection_sort	misc	150

Literaturverzeichnis

[AbsInt 2008] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers. `http://www.absint.com/ait`, Abruf: 14.08.2008 23:02.

[Allen u. Kennedy 2001] Randy Allen und Ken Kennedy. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001. ISBN 978-1558602861.

[Amiel u. Cowell 2000] Yoram Amiel und Frank Cowell. *Thinking about Inequality*. Cambridge University Press, 2000. ISBN 978-0521466967.

[Appel 2004] Andrew W. Appel. *Modern Compiler Implementation in C*. Cambridge, New York, Melbourne, Madrid, Cape Town: Cambridge University Press, 2004. ISBN 978-0521607650.

[Ayers u.a. 1997] Andrew Ayers, Richard Schooler, und Robert Gottlieb. Aggressive Inlining. *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, S. 134–145, 1997. New York: ACM. ISBN 0-89791-907-6.

[Breiman 2001] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001. Hingham: Kluwer Academic Publishers.

[Cavazos u. O'Boyle 2005] John Cavazos und Michael F. P. O'Boyle. Automatic Tuning of Inlining Heuristics. *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, S. 14, 2005. Washington: IEEE Computer Society. ISBN 1-59593-061-2.

[Chang u.a. 1992] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, und Wen mei W. Hwu. Profile-guided Automatic Inline Expansion for C Programs. *Software – Practice and Experience*, 22(5):349–369, 1992. New York: John Wiley & Sons, Inc.

[Chang u.a. 1991] Pohua P. Chang, Scott A. Mahlke, und Wen mei W. Hwu. Using Profile Information to Assist Classic Code Optimizations. *Software – Practice and Experience*, 21(12):1301–1321, 1991. New York: John Wiley & Sons, Inc.

[Cohn u. Lowney 2000] R. Cohn und P. Lowney. Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha. *Journal of Instruction Level Parallelism*, vol. 2:1–25, Mai 2000. `http://www.jilp.org/vol2`.

[Cooper u.a. 1991] Keith D. Cooper, Mary W. Hall, und Linda Torczon. An Experiment with Inline Substitution. *Software – Practice and Experience*, 21(6):581–601, 1991. New York: John Wiley & Sons, Inc.

[Cordes 2008] Daniel Cordes. Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib. Diplomarbeit, TU Dortmund, April 2008.

[Cousot u. Halbwachs 1978] Patrick Cousot und Nicolas Halbwachs. Automatic Discovery of Linear Restraints Among Variables of a Program. *POPL '78: Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, S. 84–96, 1978. New York: ACM.

[Davidson u. Jinturkar 2001] Jack W. Davidson und Sanjay Jinturkar. An Aggressive Approach to Loop Unrolling. 2001. Charlottesville: University of Virginia.

[Falk WS–2007/2008] Heiko Falk. Begleitmaterial zur Vorlesung: Compiler für Eingebettete Systeme. TU Dortmund, Wintersemester 2007/2008.

[Falk u.a. 2006] Heiko Falk, Paul Lokuciejewski, und Henrik Theiling. Design of a WCET-Aware C Compiler. *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, S. 121–126, 2006. Washington: IEEE Computer Society. ISBN 0-7803-9783-5.

[GCC 2008] Richard M. Stallman und die GCC-Entwicklergemeinde. *Using the GNU Compiler Collection – For gcc version 4.3.0*. Boston: GNU Press, 2008.

[Heise Online News 2008] Heise Online News. ITK-Branchenverband Bitkom sieht im Embedded-Bereich gute Chancen für Deutschland. Version: 23.04.2008 18:47. `http://www.heise.de/newsticker/meldung/106935`, Abruf: 18.05.2008 23:31.

[ICD 2008] Informatik Centrum Dortmund. Embedded Systems Profit Center. `http://www.icd.de/es`, Abruf: 04.08.2008 10:18.

[ICD-C 2008] Informatik Centrum Dortmund. *ICD-C Compiler framework - Developer Manual*. Informatik Centrum Dortmund, März 2008.

[ICD-LLIR 2008] Informatik Centrum Dortmund. *ICD Low Level Intermediate Representation backend infrastructure (LLIR) - Developer Manual*. Informatik Centrum Dortmund, März 2008.

[ISO/IEC 9899:1999 (E)] ISO/IEC 9899:1999 (E). Programming languages – C. Second edition. Schweiz: ISO/IEC, 1999.

[Kern-Isberner WS–2006/2007] Gabriele Kern-Isberner. Begleitmaterial zur Vorlesung: Darstellung, Verarbeitung und Erwerb von Wissen. TU Dortmund, Wintersemester 2006/2007.

[Kern-Isberner u. Beierle 2006] Gabriele Kern-Isberner und Christoph Beierle. *Methoden wissensbasierter Systeme. Grundlagen, Algorithmen, Anwendungen*. Wiesbaden: Friedr. Vieweg & Sohn Verlagsgesellschaft, 2006. ISBN 978-3834800107.

[Koseki u.a. 1997] A. Koseki, H. Komastu, und Y. Fukazawa. A Method for Estimating Optimal Unrolling Times for Nested Loops. *ISPAN '97: Proceedings of the 1997 International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN '97)*, S. 376, 1997. Washington: IEEE Computer Society. ISBN 0-8186-8259-0.

[Lee 1998] Corinna G. Lee. UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, Abruf: 13.09.2008 22:57.

[Lee u.a. 1997] Chunho Lee, Miodrag Potkonjak, und William H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communicatons Systems. *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, S. 330–335, 1997. Washington: IEEE Computer Society. ISBN 0-8186-7977-8.

[Lokuciejewski u.a. 2008a] Paul Lokuciejewski, Heiko Falk, und Peter Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. *ECRTS*, 0:321–330, Juli 2008. Los Alamitos: IEEE Computer Society.

[Lokuciejewski u.a. 2008b] Paul Lokuciejewski, Heiko Falk, Peter Marwedel, und Henrik Theiling. WCET-Driven, Code-Size Critical Procedure Cloning. *SCOPES '08: Proceedings of the 11th international workshop on software & compilers for embedded systems*, S. 21–30, März 2008. Munich, Germany.

[Lokuciejewski 2005] Paul Lokuciejewski. Design and Realization of Concepts for WCET Compiler Optimization. Diplomarbeit, TU Dortmund, Dezember 2005.

[LS8-Software 2008] TU Dortmund. Am LS8 entwickelte Software. `http://www-ai.cs.uni-dortmund.de/auto?expr=Software`, Abruf: 02.09.2008 12:56.

[Marwedel 2007] Peter Marwedel. *Eingebettete Systeme*. Berlin, Heidelberg, New York: Springer-Verlag, 2007. ISBN 978-3-540-34048-5.

[Memik u.a. 2001] Gokhan Memik, William H. Mangione-Smith, und Wendong Hu. NetBench: A Benchmarking Suite for Network Processors. *ICCAD '01: Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, S. 39–42, 2001. Piscataway: IEEE Computer Society. ISBN 0-7803-7249-2.

[Monsifrot u.a. 2002] Antoine Monsifrot, François Bodin, und Rene Quiniou. A Machine Learning Approach to Automatic Production of Compiler Heuristics. *AIMSA '02: Proceedings of the 10th International Conference on Artificial Intelligence: Methodology, Systems, and Applications*, S. 41–50, 2002. London: Springer-Verlag. ISBN 3-540-44127-1.

[Morik u. Ligges SS–2007] Katharina Morik und Uwe Ligges. Begleitmaterial zur Vorlesung: Wissensentdeckung in Datenbanken / Data Mining. TU Dortmund, Sommersemester 2007.

[MRTC 2008] Mälardalen Real-Time Research Center (MRTC). Mälardalen WCET Research Group – WCET Benchmarks. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`, Abruf: 21.08.2008 17:09.

[Puaut 2006] Isabelle Puaut. WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. *ECRTS '06: Proceedings of the 18th Euromicro Conference on Real-Time Systems*, S. 217–226, 2006. Washington: IEEE Computer Society. ISBN 0-7695-2619-5.

[R Project 2008] R Project. The R Project for Statistical Computing. `http://www.r-project.org`, Abruf: 02.09.2008 14:43.

[RapidMiner 2008] Rapid-I. Open-Source Data Mining mit der Java Software RapidMiner. `http://rapid-i.com`, Abruf: 02.09.2008 13:00.

[Schmaranz 2001] Klaus Schmaranz. *Softwareentwicklung in C*. Berlin, Heidelberg, New York: Springer-Verlag, 2001. ISBN 978-3540419587.

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

[Schwarzer 2007] Martin Schwarzer. Untersuchung des Einflusses von Compiler-Optimierungen auf die maximale Programm-Laufzeit (WCET). Diplomarbeit, TU Dortmund, Januar 2007.

[Stephenson u. Amarasinghe 2005] Mark Stephenson und Saman Amarasinghe. Predicting Unroll Factors Using Supervised Classification. *CGO '05: Proceedings of the international symposium on Code generation and optimization*, S. 123–134, 2005. Washington: IEEE Computer Society. ISBN 0-7695-2298-X.

[Witten u. Frank 2001] Ian H. Witten und Eibe Frank. *Data Mining. Praktische Werkzeuge und Techniken für das maschinelle Lernen*. München, Wien: Carl Hanser Verlag, 2001. ISBN 978-3446215337.

[Zhao u.a. 2006] Wankang Zhao, William Kreahling, David Whalley, Christopher Healy, und Frank Mueller. Improving WCET by Applying Worst-Case Path Optimizations. *Real-Time Syst.*, 34(2):129–152, 2006. Norwell: Kluwer Academic Publishers.

[Zhao u.a. 2004] Wankang Zhao, David Whalley, Christopher Healy, und Frank Mueller. WCET Code Positioning. *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, S. 81–91, 2004. Washington: IEEE Computer Society. ISBN 0-7695-2247-5.

[Živojnović u.a. 1994] Vojin Živojnović, Juan M. Velarde, Christian Schläger, und Heinrich Meyr. DSPstone: A DSP-oriented Benchmarking Methodology. *Proceedings of the International Conference on Signal Processing and Technology (ICSPAT'94)*, 1994. Dallas.

Stichwortverzeichnis