TECHNISCHE UNIVERSITÄT
DORTMUND
FAKULTÄT FÜR INFORMATIK

technische universität
dortmund

Timon Kelter

Superblock-basierte High-Level WCET-Optimierungen

Diplomarbeit

4. September 2009

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl Informatik XII
Arbeitsgruppe
"Entwurfsautomatisierung für Eingebettete Systeme"

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

GERMANY · D-44221 DORTMUND

Vorwort

An dieser Stelle möchte ich mich bei allen, die mich beim Erstellen dieser Diplomarbeit unterstützt haben, bedanken, insbesondere bei den Mitarbeitern des Lehrstuhls 12, ohne deren langjährige Entwicklungsarbeit am WCC diese Diplomarbeit unmöglich gewesen wäre. Insbesondere zu Dank verpflichtet bin ich meinem Betreuer Paul Lokuciejewski und Sascha Plazar, die mich bei auftauchenden Problemen immer tatkräftig unterstützt haben.

Ein großer Dank gebührt auch meiner Freundin Sarah und meinen Bekannten und Freunden, die während der Diplomarbeit stets zu mir gehalten haben, obwohl ich oft nur wenig Zeit für sie übrig hatte.

Schließlich danke ich auch meiner Familie für den Rückhalt, den sie mir während meiner gesamten Studienzeit gegeben hat.

Inhaltsverzeichnis

1. Einleitung

Diese Einführung stellt die Motivation (Abschnitt 1.1) und die Ziele (Abschnitt 1.2) der Diplomarbeit vor, ebenso wie eine Übersicht über verwandte Arbeiten (Abschnitt 1.3) und den weiteren Aufbau (Abschnitt 1.4) der Diplomarbeit.

1.1. Motivation

Die Verbreitung und Weiterentwicklung der digitalen Systeme haben in den letzten Jahrzehnten das Leben und die Gesellschaft stark beeinflusst. Nach der Ablösung der Großrechner der 60er und 70er Jahre durch den PC wird die momentan stattfindende Transformation hin zu ***ubiquitären Systemen*** als 3. große Entwicklungsstufe der digitalen Systeme betrachtet. Diese ***ubiquitären*** oder ***eingebetteten Systeme*** sind gekennzeichnet durch eine Einbettung in einen größeren Kontext, in dem für den Benutzer nicht mehr die Funktion als Rechner im Vordergrund steht, sondern eine konkrete Anwendung ([Marwedel, 2007]). Der Begriff der ubiquitären Systeme wurde geprägt durch Mark Weisers bekanntes Papier "The computer for the 21st century" [Weiser, 1991], in dem der Charakter solcher Systeme besonders gut umschrieben wurde: "The most profound technologies are those that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it."

Praxisbeispiele für solche eingebetteten Systeme finden sich in diversen Anwendungen, angefangen beim digitalen Fernsehen [Sieber, 2006] über moderne Automobile (ESP, ABS), die mittlerweile schon Programme mit über einer Million Zeilen Quellcode beinhalten [Keller, 2007], bis hin zu intelligenter Steuerungs- und Regelungstechnik für den Hausgebrauch [Steinle & Halang, 2006].

Die Zielsetzungen bei der Entwicklung dieser eingebetteten Systeme sind oft sehr verschieden von denen beim Entwurf klassischer Softwaresysteme. Während bei letzteren meist Speicherkapazität und Rechenleistung in mehr als ausreichendem Maß vorhanden sind, können diese bei eingebetteten Systemen stark beschränkt sein. Weitere Kriterien wie z.B. der Energieverbrauch, das Gewicht oder die Einhaltung harter Echtzeitschranken spielen bei klassischen Softwaresystemen so gut wie gar keine Rolle. Gerade der letzte Punkt spielt bei der Entwicklung von sicherheitskritischer, eingebetteter Software eine elementare Rolle, wo das verspätete Abliefern eines Ergebnisses gleichbedeutend mit dem Scheitern der gesamten Berechnung ist oder sogar katastrophale Folgen verursachen kann, so wie z.B. bei der Steuerung von ABS, ESP oder Airbags, um einige klassische Beispiele zu nennen. Solche Systeme bezeichnet man deswegen als ***harte Real-*** oder ***Echtzeitsysteme***, während Systeme, bei denen das Einhalten der Zeitschranken zwar wünschenswert, aber nicht garantiert sein muß, als ***weiche Echtzeitsysteme*** bezeichnet werden. Typische Beispiele für harte Echtzeitsysteme sind die oben genannten sicherheitskritischen Anwendungen im Auto sowie Anwendungen in der Luft- und Raumfahrt wie z.B. ein Autopilot. Andere Beispiele wie Lenkassistenz für Fahrzeuge mit Anhängern [Zöbel et al., 2006] und ***Mixed-Reality*** Anwendungen [P.F. Elzer, 2006] befinden sich je nach Anwendungsfall auf der Grenze zwischen harten und weichen Echtzeitsystemen, während z.B. Multimedia-Dienste üblicherweise als weiche Echtzeitsysteme klassifiziert werden.

Um diese harten Echtzeitschranken einzuhalten, muß die maximale Laufzeit eines Programms bekannt sein, die als ***WCET*** (für ***Worst-Case Execution Time***) bezeichnet wird. Diese WCET

ist jedoch für nichttriviale Programme nicht mehr manuell bestimmbar, weil sie von vielen Aspekten der verwendeten Hardwareplattform (Pipelining, Superskalarität, Cacheverhalten, Branch Prediction) einerseits und dem verwendeten Programm (Komplexität der verwendeten Algorithmen, verwendete Operationen) sowie dessen Eingabe andererseits abhängt. Um diese Schranken einzuhalten, wurden in der Vergangenheit häufig im Endprodukt mehr oder schnellere Hardwareressourcen eingesetzt als im Test während der Entwicklung ermittelt. Dieses Vorgehen ist mit der Hoffnung verbunden, daß diese Hardware dann schnell genug ist, um auch in ungetesteten Fällen schnell genug zu reagieren, liefert aber natürlich keine Garantie dafür. Aus diesem Grund wurden in den vergangenen Jahren an verschiedenen Instituten und Forschungseinrichtungen Methoden zur statischen Analyse der WCET entwickelt, die eine sichere obere Grenze für die WCET eines Programms liefern. Dadurch können auch unnötige Überdimensionierungen der Hardware, wie oben beschrieben, vermieden werden.

Eines dieser Analysewerkzeuge, ***aiT*** [AbsInt, 2009] von der Firma AbsInt, wurde am Lehrstuhl 12 der Technischen Universität Dortmund in einen neu entwickelten Compiler integriert, so daß die automatische Optimierung der WCET möglich wird.

1.2. Ziele der Diplomarbeit

Dieser Compiler, der ***WCC*** (WCET-aware C Compiler), wurde in den letzten Jahren am Lehrstuhl 12 entwickelt ([Falk et al., 2006], [Lokuciejewski, 2007]) und integriert neben der automatischen WCET-Analyse mit dem externen Werkzeug ***aiT*** weitere vielfältige Funktionen, wie z.B. die komfortable Annotation und Weiterverarbeitung von aiT-Eingabedaten (Schleifengrenzen) im C Quelltext [Schulte, 2007], deren halbautomatische Berechnung [Cordes, 2008], automatische Scratchpad-Allokationen [Falk & Kleinsorge, 2009, Rotthowe, 2008] und WCET-gesteuerte Optimierungen wie z.B. Registerallokation [Schmoll, 2008], Function Inlining [Lokuciejewski et al., 2009c], Loop Unrolling [Lokuciejewski & Marwedel, 2009], Loop Nest Splitting [Heiko Falk, 2006] und Procedure Cloning / Function Specialization [Lokuciejewski et al., 2008]. Diese WCET-basierten Optimierungen ermöglichen es erstmals, die WCET eines Programms vollautomatisch zu minimieren und damit frühere, kosten- oder zeitintensivere Methoden, wie z.B. die manuelle WCET-Minimierung oder die Überdimensionierung der Hardware, zu vermeiden. In einigen Fällen ist eine manuelle WCET-Minimierung mittlerweile auch ausgeschlossen, da die Entwicklung vollständig werkzeuggesteuert abläuft. So generiert z.B. das Programm Statemate [Telelogic, 2009] aus einem High-Level Zustandsübergangsdiagramm automatisch den Quellcode. Ähnliche Funktionalität bieten die Werkzeuge ASCET [ETAS, 2009] und SCADE [Esterel, 2009]. Gerade hier bietet sich ein großes Potential für die automatische WCET-Minimierung.

Das zentrale Ziel dieser Diplomarbeit ist die Entwicklung einer Gruppe neuer WCET-gesteuerter Optimierungen, die auf dem aus dem ***Profiling*** bekannten Optimierungskonzept der ***Superblöcke*** aufbauen, dieses aber auf die Anwendung zur WCET-Minimierung auf der High-Level Darstellung des C-Code übertragen. Im einzelnen sind dabei folgende Anforderungen gegeben:

- ***Traceselektion***: Es muß eine Möglichkeit geschaffen werden, auf Basis der im WCC verfügbaren WCET-Daten Programmpfade aufzuwählen, die für die Superblockbildung verwendet werden sollen.
- ***Superblockbildung***: Diese Programmpfade sollen auf der High-Level Ebene in Superblöcke transformiert werden. Hierzu ist das Konzept der Superblöcke und der Algorithmus zu ihrer Erzeugung in geeigneter Weise auf die High-Level Ebene zu übertragen.
- Die in der ICD-C vorhandene Optimierung ***Common Subexpression Elimination*** soll für die Verwendung in Superblöcken adaptiert werden (***Superblock-CSE***). Hierbei sind nach

Möglichkeit die Ergebnisse einer im WCC vorhandenen Alias-Analyse zu integrieren.

- Die ***Dead Code Elimination*** bzw. ***Operation Migration*** (s. [Chang et al., 1991]) soll für Superblöcke implementiert werden (***Superblock-DCE***). Auch hier sollten nach Möglichkeit die Ergebnisse der Alias-Analyse mit verwertet werden.
- Zur Erreichung der genannten Ziele ist es nötig, einige weitere Komponenten zu implementieren:
  - Eine Analyse zur schnellen Neuberechnung des WCEP
  - Die Berechnung von High-Level Def/Use Sets
  - Eine High-Level Liveness-Analyse
  - Die im WCC bereits vorhandene ***Back-Annotation*** (Transformation von WCET-Informationen aus der Low-Level Darstellung in die High-Level Darstellung) muß an einigen Stellen ergänzt werden

Die High-Level Darstellung wurde gewählt, um größere Synergieeffekte mit anderen Optimierungen zu ermöglichen. Da die Superblockbildung prinzipiell in der Lage ist Datenabhängigkeiten im gegebenen Programm zu verringern, können andere Optimierungen von dieser größeren Flexibilität profitieren. Durch die Wahl der High-Level Darstellung ist die Menge der Optimierungen größer, die in der Optimierungssequenz auf die Superblockbildung folgen, und die damit die Möglichkeit haben die potentiell gesteigerte Flexibilität zu nutzen.

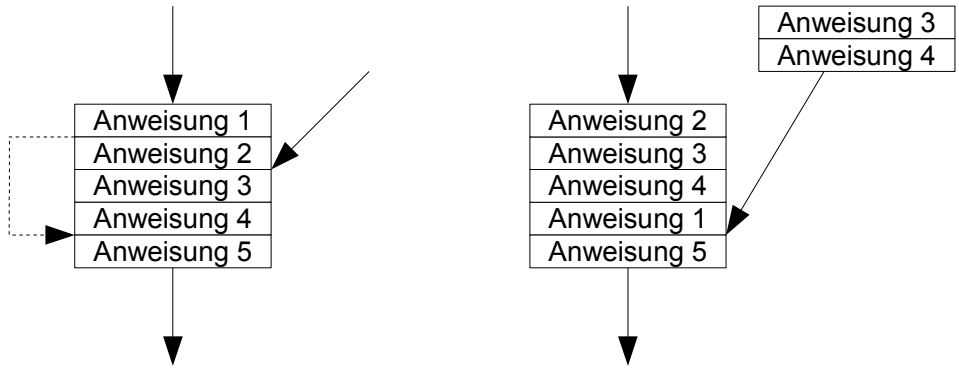
Dem Autor sind keine anderen Werke bekannt, die die Superblockbildung auf die Anwendung in der hochsprachlichen Ebene übertragen haben.

1.3. Verwandte Arbeiten

1.3.1. Profiling-basierte Arbeiten

Das Konzept der ***Superblöcke*** wurde zum ersten Mal in [Chang et al., 1991] erwähnt, aufbauend auf dem wesentlich älteren ***Trace-Scheduling*** von Fisher [Fisher, 1981]. Die Idee der Low-Level Optimierung "Trace-Scheduling" war es, einzelne, schleifenfreie und besonders häufig ausgeführte Programmpfade auszuwählen (als ***Traces*** bezeichnet) und auf diesen Programmabschnitten völlig losgelöst vom Rest des Programms ein ***Scheduling*** durchzuführen. Die Traces wurden aufgrund von empirisch ermittelten Ausführungshäufigkeiten (pro Basisblock oder pro Kontrollflußkante) ausgewählt. Die Ermittlung dieser Häufigkeiten wird dabei als ***Profiling*** bezeichnet. Da sie Ergebnisse von konkreten Versuchsläufen des Programms zusammenfaßt, ist sie nur für die Minimierung der ***ACET*** (***Average-Case Execution Time***) geeignet und vorgesehen gewesen. Ein weiteres Problem der Profiling-basierten Optimierungen wie dem Trace-Scheduling besteht darin, daß die Informationen, aufgrund derer Optimierungsentscheidungen getroffen werden, bei neuen Eingabedaten ungültig werden können.
Das Trace-Scheduling ist in [Su et al., 1984] weiter verfeinert worden, um die Optimierungszeit und den zusätzlich verbrauchten Speicherplatz zu beschränken. [Chang & Hwu, 1988] untersuchte die Auswirkungen verschiedener Traceselektionsverfahren auf die Performance der Optimierung und Lowney integrierte das Trace-Scheduling zum ersten Mal in einen kommerziellen Compiler [Lowney et al., 1993].

Dadurch, daß beim Trace-Scheduling Anweisungen über die Grenzen von Basisblöcken hinaus verschoben wurden, gab es eine Reihe von Spezialfällen, in denen ***Kompensationscode*** eingefügt werden musste, um die Programmsemantik zu wahren. Als Beispiel dient die Verschiebung von Anweisung 1 in Abbildung 1.1. Diese Fälle traten insbesondere bei der Verschiebung von An-



(a) Das ursprüngliche Codesegment

(b) Code nach der Verschiebung von Anweisung 1

Abbildung 1.1.: Kompensationscodeerzeugung beim Trace-Scheduling.

weisungen über Einsprungpunkte im Trace auf (siehe Abbildung 1.1(b)). Um dieses Problem zu vermeiden, entwickelte Chang [Chang et al., 1991] das Konzept des Superblocks, der diese Fälle durch geeignete Codetransformationen eliminierte. In diesem Papier wurden zugleich die ersten ***Superblock***-spezifischen Optimierungen vorgestellt. Das Konzept wurde danach in vielen Fällen als das praktikablere von beiden wieder aufgegriffen, z.B. von Cohn [Cohn & Lowney, 2000], der eine Verallgemeinerung der ***Superblock Loop Optimizations*** aus [Chang et al., 1991] einführte, von Hwu [Hwu et al., 1993], der ***Superblock Extension Optimizations*** analysierte, und von Chen [Chen et al., 1992], der ***Software-Pipelining*** und die Auflösung von Speicherzugriffskonflikten für Superblöcke intensiver behandelte.

1.3.2. WCET-basierte Arbeiten

Zum Thema WCET-Optimierung sind bisher schon verschiedenste Arbeiten veröffentlicht worden, unter anderen wurden WCET-sensitive Versionen vieler klassischer Optimierungen vorgestellt, so z.B. die in Abschnitt 1.2 erwähnten Optimierungen des Lehrstuhls 12. Zhao [Prasad et al., 2004] versuchte, WCET-optimierende Sequenzen klassischer Optimierungsalgorithmen mithilfe von genetischen Algorithmen zu finden, und Puaut [Puaut, 2006] präsentierte einen Algorithmus zum statischen Instruction-Cache-Locking mit dem Ziel der WCET-Minimierung. Die einzigen dem Autor bekannten High-Level WCET-Optimierungen wurden am Lehrstuhl 12 entwickelt. Im einzelnen handelt es sich um

- Function Inlining [Lokuciejewski et al., 2009c],
- Loop Unrolling [Lokuciejewski & Marwedel, 2009],
- Loop Nest Splitting [Heiko Falk, 2006],
- Procedure Cloning / Function Specialization [Lokuciejewski et al., 2008] und
- Loop Unswitching [Lokuciejewski et al., 2009b].

Es handelt sich hierbei um allgemein bekannte High-Level Optimierungen, die jedoch in den jeweiligen Arbeiten speziell auf die WCET-gesteuerte Anwendung zugeschnitten wurden. Die Übertragung der Superblock-Optimierung auf die WCET-Optimierung wurde von Zhao in [Zhao et al., 2006] bereits für den Low-Level Fall versucht. Die vorliegende Arbeit unterscheidet sich von der genannten allerdings in einigen wesentlichen Punkten. Anders als in [Zhao et al., 2006] wurde

- die Superblockbildung als High-Level Optimierung durchgeführt. Dadurch findet sie wesentlich früher in der Optimierungskette statt und kann somit neues Potential für andere Optimierungen freisetzen. Diese Strategie, Optimierungen, die intensive Änderungen am Kontrollfluß vornehmen, möglichst früh auszuführen, ist auch bekannt als ***Structural Compilation*** und wird z.B. auch im GCC angewandt ([Kidd & Hwu, 2006]).

- nicht nur die Superblockbildung selbst implementiert, sondern weitere speziell auf Superblöcke ausgelegte Optimierungen wie die Superblock-CSE und Superblock-DCE.

- das Phänomen der Pfadwechsel, das Zhao durch eine Neuberechnung mit dem verwendeten WCET Analyzer löst, gesondert betrachtet. Hierfür wurde eine spezielle Bibliothek implementiert (siehe Kapitel 5), da das Verfahren von Zhao bei großen Programmen schnell an seine (laufzeitbedingten) Grenzen stößt.

- die Vermeidung von Sprüngen bei uns weniger stark in den Vordergrund gerückt (Die in [Zhao et al., 2006] verwendete Architektur weist sehr hohe Branch Penalties auf und macht damit eine Fokussierung auf Sprungvermeidung nötig.)

- ein auch in der Industrie eingesetztes, IPET-basiertes WCET-Analyseverfahren benutzt, in [Zhao et al., 2006] wird ein pfadbasiertes Verfahren benutzt (siehe Kapitel 2)

- eine umfangreiche Sammlung von größeren Realworld-Benchmarks zur Auswertung benutzt. Die durchschnittliche Größe der Beispiele aus [Zhao et al., 2006] beträgt nur 150 Zeilen, dadurch sind die Ergebnisse in Bezug auf reale Anwendungen wenig aussagekräftig.

1.4. Aufbau der Diplomarbeit

In **Kapitel 2** wird zuerst die WCET formal definiert, und die heute verfügbaren Methoden der WCET-Analyse sowie ihre Vor- sowie Nachteile werden vorgestellt.

Danach wird in **Kapitel 3** das am Lehrstuhl 12 verwendete Compiler-Framework des WCC und der dabei verwendete TriCore-Prozessor vorgestellt. Es wird geschildert, inwieweit die bestehenden Abläufe erweitert werden mußten, um die in dieser Arbeit behandelten Optimierungen zu implementieren.

Das **Kapitel 4** geht dann auf die Definition und den Nutzen der Superblöcke ein und erläutert kurz deren Entwicklung sowie die Besonderheiten der hier verwendeten High-Level-Superblöcke. Außerdem werden die bekannten Superblock-spezifischen Optimierungen dargestellt.

Die Neuberechnung des ***WCEP*** (***Worst Case Execution Time Paths***) wird in **Kapitel 5** thematisiert. Sie ist bei iterativen Optimierungen wie der Superblockbildung ein unverzichtbares Hilfsmittel und wird deshalb hier zuerst vorgestellt.

In **Kapitel 6** werden die in dieser Arbeit verwendeten Algorithmen vorgestellt, mit denen die Erzeugung von High-Level-Superblöcken realisiert wird. Diese gliedern sich in einen vorgeschalteten Prepass, der vorbereitende Code-Transformationen durchführt, eine Traceselektion zur Identifizierung der zu transformierenden Codepfade und die Transformierung dieser Pfade in Superblöcke. Zusätzlich werden zwei Techniken vorgestellt, mit denen das Codewachstum begrenzt und im Idealfall auch die WCET verringert werden kann.

Kapitel 7 stellt die beiden implementierten Superblock-Optimierungen, Superblock-CSE und Superblock-DCE, vor und präsentiert auch die dafür benötigten Analysen, soweit sie im WCC

bisher noch nicht vorhanden waren.

Schlußendlich werden in **Kapitel 8** die experimentellen Ergebnisse der implementierten Optimierungen dargestellt und **Kapitel 9** gibt eine Zusammenfassung des Erreichten sowie einen kurzen Ausblick über mögliche Folgearbeiten.

2. WCET-Analyse

Die möglichst präzise Schätzung der WCET und des WCEP eines gegebenen Programms ist eine hochkomplexe Aufgabe, die für die Optimierung der WCET absolut unerläßlich ist. Der WCET-Begriff wird im folgenden definiert (Abschnitt 2.2) und ein kurzer Überblick über die Vielzahl von existierenden Lösungsansätzen wird gegeben (Abschnitte 2.3, 2.4, 2.5), da das im WCC verwendete Analysewerkzeug aiT auf einem dieser Ansätze aufbaut und wir diesem Ansatz auch selbst nutzen werden, um den WCEP mit geringem Rechenaufwand bei Bedarf neu zu berechnen (siehe Kapitel 5). Zum Schluß wird der Aufbau und die Funktionsweise von aiT skizziert (Abschnitt 2.6).

2.1. Der Kontrollflußgraph

Wähend die WCET nur ein Zahlenwert ist, stellt der WCEP einen Pfad durch das Programm dar, so daß hier zuerst der Pfadbegriff definiert werden soll.

Der ***Kontrollflußgraph*** (CFG, von ***Control Flow Graph***) eines Programms P ist ein gerichteter Graph $G = (V, E)$ wobei V die Menge der Basisblöcke und E die Menge der Kontrollflußkanten ist. Dabei ist ein ***Basisblock*** eine maximale Menge von direkt aufeinanderfolgenden Anweisungen, die im Programmablauf nur an der ersten Anweisung betreten und nur an der letzten wieder verlassen werden können. Der Basisblock, an dem die Startfunktion (üblicherweise die `main`-Funktion) betreten wird, wird als ***Quelle*** bezeichnet, und alle Basisblöcke, an denen die Startfunktion wieder verlassen werden kann, werden als ***Senken*** bezeichnet. Eine ***Kontrollflußkante*** ist immer dann zwischen 2 Basisblöcken b_i und b_j vorhanden, wenn der Kontrollfluß nach Ausführung von b_i nach b_j wechseln kann. Abbildung 2.1 zeigt ein Beispiel für einen Kontrollflußgraph, die Quelle ist dort hellgrau, die Senke dunkelgrau markiert. Für die Menge der Kontrollflußkanten, die einen Knoten verlassen oder betreten, verwenden wir die Kürzel $\delta^+(b) = \{(b_{start}, b_{end}) \in E | b_{start} = b\}$ bzw. $\delta^-(b) = \{(b_{start}, b_{end}) \in E | b_{end} = b\}$. Ein ***Pfad*** ist eine Liste von Basisblöcken $P = (b_1, b_2, ..., b_k)$, so daß für alle b_i gilt, daß $(b_i, b_{i+1}) \in E$ ist. Die ***Laufzeit*** eines Pfades wird dabei durch die Anzahl der verstrichenen Prozessorzyklen auf der verwendeten Hardwareplattform angegeben, die bei

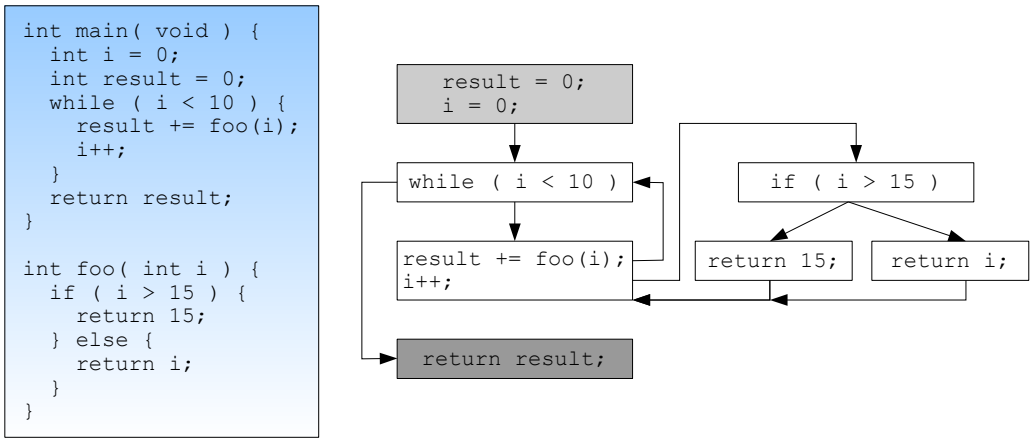


Abbildung 2.1.: Beispiel für einen Kontrollflußgraph über 2 Funktionen.

Abbildung 2.2.: Illustration einer sicheren $WCET_{est}$.

einmaliger Ausführung dieses Pfades anfallen. Als ***Programmpfad*** bezeichnen wir einen Pfad von der Quelle zu einer beliebigen Senke, also einen Pfad durch das gesamte Programm. Insbesondere sind in dieser Definition auch Pfade enthalten, die zwar als Folge miteinander verbundener Knoten im Kontrollflußgraphen existieren, die aber als reale Anweisungsfolge gar nicht ausführbar sind, z.B. aufgrund von sich widersprechenden `if`-Bedingungen. Solche Pfade werden als ***infeasible*** bzw. ***unausführbar*** bezeichnet. Aufbauend darauf wird ein Basisblock b genau dann als ***unausführbar*** bezeichnet, wenn alle Programmpfade p mit $b \in p$ unausführbar sind. Die Anweisungen in einem solchen Basisblock bezeichnen wir als ***Dead Code***. Je nach verwendetem Analyseverfahren ist es eine mehr oder weniger große Herausforderung, unausführbare Pfade von der WCET-Analyse auszuschließen. Dies ist wünschenswert, da es ansonsten zu Überschätzungen kommen kann. Dead Code hingegen kann oftmals relativ einfach eliminiert werden.

Die genannten Konzepte sind sowohl auf der High-Level als auch auf der Low-Level Darstellung eines Programms anwendbar, allerdings mit einigen Unterschieden im Detail, siehe dazu den Abschnitt über die ***Backannotation*** in 3.1.

2.2. WCET-Begriff

Wie bereits angeklungen unterscheidet man zwischen der geschätzten $WCET_{est}$ und der echten $WCET_{real}$. Als ***WCEP*** bezeichnen wir den Programmpfad, der unter allen Programmpfaden, bei Betrachtung aller möglichen Eingaben, die maximale Laufzeit aufweist. Diese Laufzeit bezeichnen wir als $WCET_{real}$. Die $WCET_{est}$ ist eine sichere obere Schranke, so daß jederzeit

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

gilt (siehe Abbildung 2.2). In diesem Fall bezeichnet man die ermittelte $WCET_{est}$ als ***sicher***.
Diese Unterscheidung zwischen $WCET_{real}$ und $WCET_{est}$ ist nötig, da die exakte Berechnung der maximalen Laufzeit im allgemeinen Fall unentscheidbar ist. Dies läßt sich leicht zeigen, denn die exakte Berechnung der $WCET_{real}$ würde sofort eine Lösung für das Halteproblem liefern, von dem Alan Turing bereits 1936 [Wegener, 1999, S.22] bewies, daß es unentscheidbar ist.
Bei der Bestimmung der $WCET_{est}$ soll also die Schätzung sowohl sicher, als auch möglichst ***präzise*** sein, d.h. $WCET_{est} - WCET_{real}$ soll gegen 0 gehen. In der Praxis stehen meist nur die $WCET_{est}$-Werte zur Verfügung, während die $WCET_{real}$-Werte im allgemeinen unbekannt sind. Im folgenden verwenden wir daher die Bezeichnung WCET als Kurzform für die $WCET_{est}$.

2.3. Dynamische Methoden

Bei der dynamischen Bestimmung der WCET handelt es sich um experimentelle Methoden, d.h. es wird versucht, die WCET des Programms durch Messungen bei Probeläufen mit passenden Eingaben zu bestimmen. Der große Vorteil hierbei ist, daß keine weiteren Modellierungs- oder Berechnungsschritte benötigt werden, sondern nur das gegebene Programm und die Zielplattform. Praktische Relevanz hat dieses Verfahren daher vor allem durch seine einfache Umsetzung erlangt.

Dabei können für realistische Programme aber nicht alle, sondern nur ausgewählte Eingaben getestet werden, um die Laufzeit der Analyse zu beschränken. Auch wenn verschiedene Strategien bei der Auswahl der Eingaben möglich sind, wie z.B. die Verwendung von Randomisierung oder evolutionären Algorithmen, so sind die erzielten Ergebnisse doch nicht sicher im Sinne von Gleichung 2.1, denn für sie gilt lediglich $WCET_{est} \leq WCET_{real}$. Daher wird dieses Verfahren im folgenden nicht mehr betrachtet.

2.4. Statische Methoden

Im Gegensatz dazu versuchen statische Methoden, die WCET zu bestimmen, ohne auf Daten aus Probeläufen des Programms zurückzugreifen. Dazu wird ein abstraktes Modell der zugrundeliegenden Plattform benötigt, auf dem man die Laufzeiten der möglichen Pfade durch das Programm analysieren kann. Im Grunde sind hier 2 Teilprobleme zu lösen, nämlich die Bestimmung des WCEP unter allen Programmpfaden und die Berechnung der auf ihm anfallenden WCET. Es existieren 3 bekanntere Ansätze, die im folgenden vorgestellt werden. Alle haben gemeinsam, daß sie im allgemeinen Fall nicht ohne Benutzerhilfe auskommen, da das zugrundeliegende Problem wie bereits erwähnt unentscheidbar ist. So müssen bei allen Verfahren für komplexere Programme Schleifengrenzen oder Rekursionsschranken an das Analysewerkzeug übergeben werden, damit ein Ergebnis erzeugt werden kann. Diese Zusatzinformationen werden ***Flow Facts*** genannt. Die Bestimmung dieser Schleifengrenzen kann durch eine Schleifenanalyse (wie z.B. in [Lokuciejewski et al., 2009a]) geschehen, oder falls das nicht möglich ist, durch den Benutzer selbst, der dann dafür verantwortlich ist, die Information über die Schleifengrenzen korrekt zu halten.

2.4.1. Syntaxbaumbasierte Methoden

Syntaxbaumbasierte Methoden gehören zu den ältesten der bekannten Methoden. Eine der ersten Erwähnungen finden sie z.B. bei Park & Shaw in [Park & Shaw, 1991]. Sie versuchen die WCET eines Programms über die Top-Down Anwendung von festen Regeln im Syntaxbaum zu berechnen. So wird z.B. die WCET einer `if`-Anweisung `A := if (b) { B; } else { C; }` bzw. einer `while`-Anweisung `W := while (b) { B; }` folgendermaßen abgeleitet:

$$WCET_{est}(A) = WCET_{est}(b) + \max\{WCET_{est}(B), WCET_{est}(C)\} \tag{2.2}$$

$$WCET_{est}(W) = (N+1) \cdot WCET_{est}(b) + N \cdot WCET_{est}(B) \tag{2.3}$$

wobei N eine vorher bestimmte Schleifengrenze ist. Die WCET elementarer Anweisungen erhält man dabei durch die Vorhersage des erzeugten Assemblercode und die Schätzung von dessen Laufzeit. Für die Transitionen zwischen den einzelnen Befehlen werden weitere Kosten berechnet, so daß man insgesamt die WCET einer Funktion rekursiv durch Aufaddieren der Anweisungs-WCETs und Transitionskosten erhält.

Der größte Vorteil dieses Ansatzes ist die hohe Laufzeiteffizienz, allerdings ist es hier schwieriger, weitere Informationen in die Analyse mit einfließen zu lassen, wie z.B. die Exklusion unzulässiger Ausführungspfade. Dazu kommt, daß Effekte, die erst im Maschinencode auftreten, insbesondere solche Effekte, die im Zusammenspiel mehrerer Instruktionen oder Basisblöcke auftreten, bei diesem Verfahren schwerer zu berücksichtigen sind. Verbesserungen des Ansatzes sind die Modellierung fortgeschrittener Hardwareeigenschaften wie Caches, Pipelines und Branch Prediction in [Colin & Puaut, 2001], [Lim et al., 1995], die stärkere Berücksichtigung von Pfaden im CFG durch Zusammenfassen einzelner Elemente des Syntaxbaums [Betts & Bernat, 2006] oder die automatische Bestimmung von Stellen, an denen die WCET möglicherweise überschätzt wurde [Avila et al., 2003].

Dieses Verfahren wurde bereits im Forschungscompiler HEPTANE ([Colin & Puaut, 2001]) angewandt, der am IRISA Institut im französischen Rennes entwickelt wurde. Ein zentraler Nachteil der Technik ist jedoch, daß Low-Level Optimierungen nicht mit in die WCET-Analyse einfließen, da die WCET der elementaren Anweisungen wie erwähnt aufgrund von Vorhersagen über den generierten Assemblercode abgeschätzt wird. Die Auswirkungen von Low-Level Optimierungen auf den Assemblercode sind jedoch nur mit enormem Aufwand vorhersagbar.

2.4.2. Pfadbasierte Methoden

Die pfadbasierten Methoden betrachten nicht einzelne Elemente des Syntaxbaums, sondern versuchen, die WCET zu ermitteln, indem die Laufzeit jedes einzelnen Programmpfades ermittelt wird und das Maximum gewählt wird. Diese Laufzeiten werden hierbei auch wieder über die Analyse der auf dem Pfad ausgeführten Maschinenbefehle bestimmt. Falls wirklich alle Programmpfade einzeln analysiert werden, ist dieses Verfahren sehr exakt. Da es im allgemeinen jedoch exponentiell viele Pfade geben kann (als Beispiel dient ein Programm, das ausschließlich aus einer sequentiellen Abfolge von `if`-Anweisungen besteht), ist das Worst-Case Verhalten dieser Methode sehr schlecht und auch die reale Performance ist nicht konkurrenzfähig. Daher wurden Ansätze entwickelt, die eine vollständige Enumeration aller Pfade vermeiden [Chen et al., 2007] oder die pfadbasierten Techniken werden nur als Ergänzung zur Verbesserung der Genauigkeit der Syntaxbaum- oder IPET-basierten Methoden auf ausgewählte Programmteile angewandt [Betts & Bernat, 2006].

2.4.3. IPET Methode

Die ***IPET*** Methode (für ***Implicit Path Enumeration Technique***) vereinigt das Finden des WCEP und das Bestimmen seiner Laufzeit in einem einzigen Schritt, der durch ein ***Integer Linear Program*** (***ILP***) gelöst wird. Dieses Verfahren, das zuerst von Li & Malik in [Li & Malik, 1995] betrachtet wurde, ist zwar theoretisch NP-vollständig hat aber in der Praxis eine sehr gute Performance, da die Lösungsverfahren für ILPs sehr gut erforscht sind [ILOG, 2009].

Zur Erstellung des ILP wird bei der ursprünglichen Version der IPET Methode für jeden Basisblock b eine maximale Laufzeit c_b angenommen. Berechnet werden muß nun, wie oft jeder Basisblock maximal ausgeführt werden kann (Ausführungshäufigkeit x_b). Die zu maximierende Zielfunktion ist damit $\sum_{b \in B} c_b \cdot x_b$, wobei B die Menge der Basisblöcke sei. Da sie in dieser Form unbegrenzt ist benötigt das Verfahren noch Bedingungen die ihren Wert begrenzen. Das sind:

- Bedingungen, die sicherstellen, daß die WCEC eines Knotens gleich der WCEC seiner eingehenden und ausgehenden Kanten ist,
- Bedingungen, die sicherstellen, daß die Anzahl der Schleifeniterationen bzw. Rekursionsstufen beschränkt wird,
- Bedingungen, die Funktionsaufrufe modellieren und schließlich
- eine Bedingung, die die WCEC des Startknotens der `main`-Funktion auf 1 festlegt.

Insgesamt bietet dieses Verfahren einen guten Mittelweg zwischen Präzision und Effizienz der Analyse und wird deshalb auch in dem im WCC eingesetzen Analysetool aiT benutzt, allerdings mit vielen Erweiterungen [AbsInt, 2009], [Ferdinand, 2004]. Die IPET Methode wird in Kapitel 5 noch einmal genauer vorgestellt, da wir sie dort für die schnelle Neuberechnung des WCEP nutzen werden.

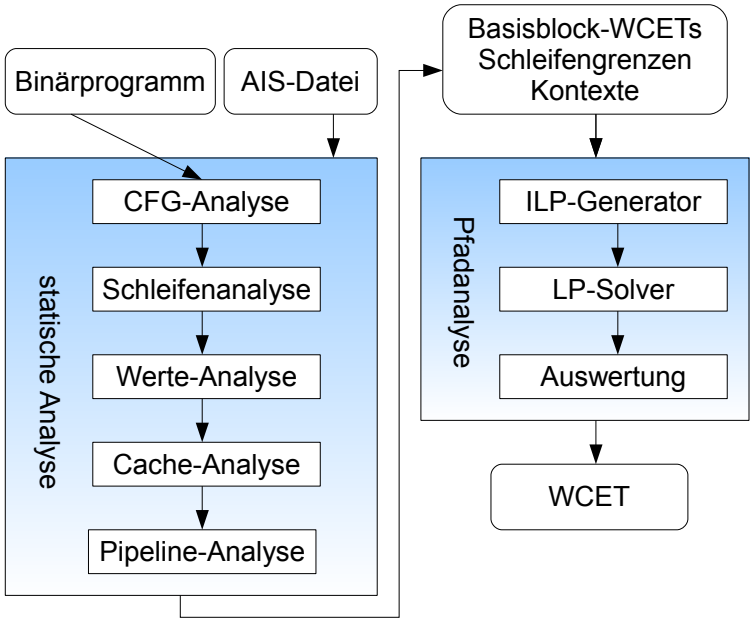


Abbildung 2.3.: Aufbau des im WCC verwendeten Analysewerkzeugs aiT.

2.5. Hybride Methoden

Hybride Methoden versuchen die Präzision der statischen Methoden, wie z.B. IPET, durch dynamische Elemente zu verbessern. Dafür wird beispielsweise die Laufzeit einzelner Pfade durch Simulation ermittelt und danach versucht, diese in das ILP des IPET-Verfahrens einzubetten. Dadurch entstehen allerdings ähnliche Probleme wie bei den rein dynamischen Verfahren, weil für den simulierten Pfad jetzt keine sichere WCET mehr vorliegt.

Ein spezielles hybrides Verfahren wird beispielsweise in SymTA/S [Symtavision, 2009] benutzt, wobei die Laufzeit verzweigungsfreier Pfade durch Simulation ermittelt wird und diese Simulationsergebnisse danach in das verwendete mathematische Modell eingebunden werden.

2.6. WCET-Analyse mit aiT

Aufbau und Funktionsweise

Wie bereits erwähnt wird im WCC das Analysetool aiT verwendet, das auch von vielen Unternehmen für die WCET-Analyse eingesetzt wird (Airbus Industries, Bosch, BMW, Nokia, IBM, Volvo) und im Vergleich zu vielen anderen Tools sehr hohe Präzision und einen großen Funktionsumfang aufweist [Wilhelm et al., 2008]. Diese Arbeitsweise von aiT soll hier anhand von Abbildung 2.6 näher erläutert werden.

Als Eingabe erwartet aiT ein Binärprogramm und Benutzerinformationen über die gewünschten Analyseoptionen, Schleifeniterationsgrenzen und Rekursionstiefen in Form einer AIS-Datei. Das Binärprogramm wird dann disassembliert und in einen Kontrollflußgraphen überführt, der dann im ***CRL2***-Format abgelegt wird. An diese Datei werden danach schrittweise weitere Informationen angehängt:

1. Die ***Schleifen-Analyse*** ermittelt Iterationsschranken für nicht vom Benutzer annotierte

Schleifen, allerdings funktioniert dies nur bei sehr einfachen Schleifen mit konstanten Iterationszahlen. Um diese Schwäche zu umgehen wurde am Lehrstuhl 12 eine komplexere Schleifenanalyse implementiert [Lokuciejewski et al., 2009a], deren Ergebnisse dann zusammen mit dem vom Benutzer angegebenen Schleifengrenzen an aiT weitergegeben werden können. Es ist aufgrund der Unberechenbarkeit des Halteproblems (s. Abschnitt 2.2) jedoch unmöglich, eine Analyse zu entwickeln, die für jede beliebige Schleife eine Iterationsschranke ermitteln kann.

2. Die ***Werte-Analyse*** versucht durch das Anwenden von Methoden der abstrakten Interpretation Wertebereiche für die in den Registern gehaltenen Inhalte festzustellen.

3. Danach versucht die ***Cache-Analyse*** das Worst-Case Verhalten beim Zugriff auf Caches vorherzusagen. Dazu werden Cache-Zugriffe in verschiedene Kategorien eingeteilt wie z.B. *always hit* und *always miss*.

4. Abschließend wird eine ***Pipeline-Analyse*** durchgeführt, die das Worst-Case Verhalten der Pipeline unter Berücksichtigung der bereits gesammelten Informationen ermittelt.

Alle aiT-Analysen arbeiten mit dem Konzept der ***Kontexte***. Ein Kontext ist dabei die Aufruf-Historie des aktuell untersuchten Basisblocks. Das hat den Nutzen, daß für denselben Basisblock in verschiedenen Kontexten (z.B. wenn die umgebende Funktion mit verschiedenen Parametern aufgerufen worden ist oder der Block in einer Schleife liegt) verschiedene Ergebnisse festgehalten werden können und man nicht auf das schlechteste Ergebnis zurückfallen muß. Mit steigender Anzahl Kontexte kann sich die Analysegenauigkeit also erhöhen, allerdings steigt mit ihr auch die Analysedauer wegen der erhöhten Komplexität. Aus diesem Grund kann der Benutzer eine maximale Anzahl Kontexte an aiT übergeben. Falls diese Anzahl Kontexte bei der Analyse überschritten wird, werden alle folgenden Analyseergebnisse im letzen Kontext aggregiert, so daß dieser Kontext als Platzhalter für alle nicht mehr einzeln verwalteten Kontexte dient. Damit wird eine reduzierte Analysedauer auf Kosten der Präzision erzwungen. Das Ergebnis der Analysen ist dann in der annotierten CRL2-Datei enthalten, unter anderen werden die maximalen Ausführungszeiten der Basisblöcke unter allen analysierten Kontexten festgehalten.

Abschließend findet die ***Pfadanalyse*** statt, die die eigentliche Berechnung der WCET des Gesamtprogramms übernimmt. Das verwendete Verfahren ist ein IPET-Verfahren, allerdings erweitert um die erwähnten Kontexte. Dafür wird das ILP bei c Kontexten c-fach vervielfältigt, bei verschachtelten Schleifen sogar noch öfter, und die Nebenbedingungen sowie die Zielfunktion werden entsprechend angepasst, so daß im ILP für jeden Block in jedem Kontext eine eigene Ausführungshäufigkeits-Variable vorhanden ist. Als Ergebnis erhält man für jeden Basisblock und jeden Kontext eine WCET und eine WCEC oder aber die Information, daß der Block unausführbar ist. Die Laufzeit des Gesamtprogramms ergibt sich dann durch Summieren über alle Basisblöcke und Kontexte.

Mit Hilfe dieser WCET-Werte können im WCC Framework, das im nächsten Kapitel vorgestellt wird, WCET-gesteuerte Optimierungen vorgenommen werden.

3. WCC (WCET-aware C Compiler)

Die in dieser Diplomarbeit implementierten Superblock-basierten Optimierungen wurden in das bestehende Compiler-Framework des WCC integriert, das deshalb im folgenden kurz vorgestellt werden soll.

Der am Lehrstuhl Informatik 12 der Technischen Universität Dortmund entwickelte WCC ist als Compiler für eingebettete Systeme konzipiert. Er übersetzt in ANSI-C (C99, [ISO/IEC, 1999]) geschriebene Programme in Maschinencode für den Infineon TriCore TC1796 bzw. TC1797. Unterstützung für weitere Hardwareplattformen wie ARM7 befindet sich momentan in der Entwicklung. Wie bereits in der Einleitung erwähnt, spielen im Bereich der eingebetteten Systeme bei der Softwareentwicklung oftmals andere Kriterien eine Rolle als in klassischen PCs. Deshalb ist es wünschenswert, daß auch die in diesem Bereich verwendeten Compiler Unterstützung für die Optimierung dieser andersartigen Kriterien zur Verfügung stellen. Die Intention bei der Entwicklung des WCC war es, erstmals einen ausgereiftes, industriell eingesetztes WCET-Analysewerkzeug in einen Compiler zu integrieren, um schon während des Übersetzungsvorgangs auf die WCET Informationen zugreifen zu können und diese z.B. für Optimierungen nutzbar zu machen.

In den folgenden Abschnitten werden zuerst der Aufbau des WCC (Abschnitt 3.1) und die zugrundeliegende TriCore-Plattform kurz vorgestellt (Abschnitt 3.2). Außerdem werden die besonderen Herausforderungen bei der Minimierung der WCET (Abschnitt 3.3) und die Bereiche, mit denen die in dieser Diplomarbeit entwickelte Optimierung am stärksten interagiert, dargestellt (Abschnitt 3.4).

3.1. Aufbau des WCC

Die grobe Struktur des WCC ist in Abbildung 3.1 dargestellt. Die schwarzen Pfeile deuten dabei den Kontrollfluß innerhalb eines normalen Compilers an, bei allen anderen Teilen handelt es sich um WCET-spezifische Ergänzungen.

Die vorhandenen Eingabedaten werden zuerst durch den ***Parser*** in eine hochsprachliche Zwischendarstellung (***intermediate representation***) mit dem Namen ***ICD-C IR*** überführt, wobei üblicherweise alle vorhandenen C-Dateien in einem Schritt verarbeitet werden, da für eine vollständige WCET-Analyse der komplette Quellcode des zu übersetzenden Programms bekannt sein muß. Die ICD-C IR (im folgenden auch kurz IR genannt) ist eine direkte Darstellung des C-Codes, so daß sich der Code bei Bedarf auch wieder ausgeben läßt, und sie bietet viele integrierte, plattformunabhängige High-Level Optimierungen und eine Unterstützung für Codeselektoren [ICD, 2008]. Der ***Codeselektor*** des WCC wurde am Lehrstuhl implementiert und basiert auf der Methode des Tree-Pattern-Matching. Seine Ausgabe ist eine Instanz der ***ICD-LLIR*** (für ***Informatik Centrum Dortmund - Low-Level Intermediate Representation***, [ICD, 2009]), die eine assemblernahe Darstellung des Programms ist. Auch in die LLIR ist wieder eine Vielzahl von generischen Optimierungen integriert, wobei Optimierungen zuerst auf der ***virtuellen LLIR***, die nur virtuelle Register benutzt, und danach auf der ***physischen LLIR*** durchgeführt werden. Die ***Registerallokation*** transformiert dabei die virtuelle in die physische LLIR. Zum Schluß überführt der ***Codegenerator*** die physische LLIR in Assemblercode und ein Linkerskript, so daß das binäre Programm durch einen Standard-Linker für den TriCore erzeugt werden kann.

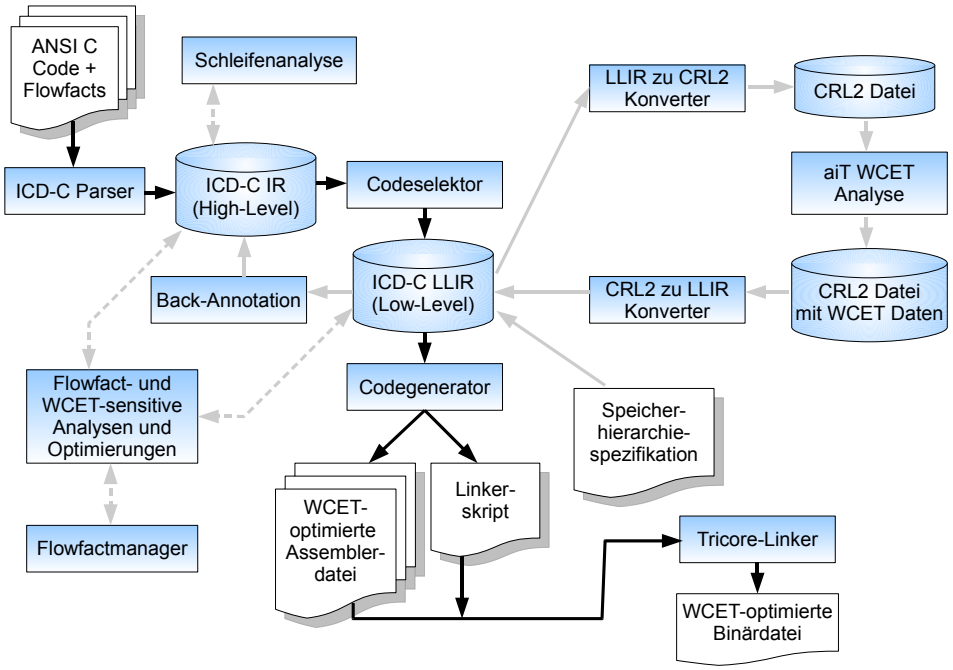


Abbildung 3.1.: Der Aufbau des am LS12 entwickelten WCET-Aware C Compiler (WCC).

Auffällig ist, daß der WCC im Gegensatz zu einigen anderen Compilern keine völlig maschinen- und hochsprachenunabhängige Zwischendarstellung besitzt. Dies ist bewußt so gestaltet worden, da die Performance des erzeugten Codes höher bewertet wird als die schnelle und einfache Portierbarkeit des Compilers auf neue Sprachen oder Plattformen.

Im folgenden sollen kurz die WCET-spezifischen Erweiterungen, die in Abbildung 3.1 durch die grauen Pfeile gekennzeichnet sind, erläutert werden. Durchgezogene graue Pfeile stehen hierbei für Funktionen, die bei einer WCET-sensitiven Kompilierung immer aktiviert sind, während gestrichelte Pfeile die Komponenten anbinden, die im Falle einer WCET-Analyse aktiviert werden können, aber nicht müssen.

3.1.1. WCET Analyse

Die physische LLIR wird vom ***LLIR→CRL2-Konverter*** in eine CRL2-Datei übersetzt, die von aiT gelesen werden kann. Damit wird der Schritt der Extraktion des Kontrollflußgraphen im normalen Arbeitsablauf von aiT übersprungen (siehe Abschnitt 2.6). Der Konverter ermöglicht die Zuordnung von LLIR-Basisblöcken zu CRL2-Basisblöcken und macht erst dadurch eine Rückübertragung der Ergebnisse in die LLIR möglich. Mit der erzeugten CRL2-Datei wird dann eine vollständige aiT-Analyse durchgeführt und deren Ergebnisse werden in einer CRL2-Datei abgelegt. Der ***CRL2→LLIR-Konverter*** überführt dann mithilfe der Basisblockzuordnung die Ergebnisse wieder in die LLIR. Die Ergebnisse werden dort als Benutzerannotationen an die LLIR-Basisblöcke angehängt und können von allen anderen Teilen des WCC abgefragt werden. Die annotierten Informationen sind im einzelnen:

- Die über alle Kontexte summierte $WCET_{sum}$
- Die über alle Kontexte summierte $WCEC_{sum}$
- Die Information, ob der Basisblock ***unausführbar*** ist (***infeasibility***)

- Die *WCEC* jeder Kante im Kontrollflußgraphen
- Die Anzahl der Instruction Cache Misses im Block

Detailliertere Informationen zur Funktionsweise der Konverter finden sich in [Lokuciejewski, 2007].

3.1.2. Back-Annotation

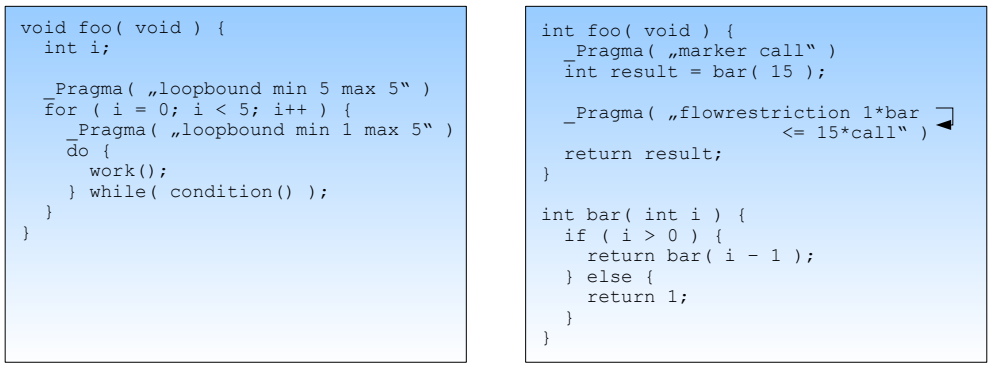
Die Back-Annotation überträgt die zuerst nur auf der LLIR-Ebene vorhandenen WCET-Informationen auf die ICD-C Ebene. Auf diese Weise wird es möglich, im WCC High-Level WCET-Optimierungen vorzunehmen. Die annotierten Werte entsprechen im wesentlichen denen auf der LLIR Ebene, nur daß IR-Basisblöcke statt LLIR-Basisblöcke annotiert werden. Außerdem werden, falls gewünscht, auch die Codegrößen der Basisblöcke in Bytes annotiert und die Anzahl der Spill-Instruktionen, die der Registerallokator pro Basisblock erzeugt hat, da diese Daten auf der LLIR Ebene ohnehin zur Verfügung stehen. Die Back-Annotation berechnet auch eine Menge von Basisblöcken, die immer auf dem WCEP liegen, die also vom Eingang der `main`-Funktion ohne Durchlaufen einer Verzweigung, die zu Nicht-WCEP-Blöcken verzweigt, erreicht werden. Diese Blöcke werden als ***Invariant Path*** bezeichnet [Lokuciejewski et al., 2009b]. Auf einige Besonderheiten der Back-Annotation werden wir in Abschnitt 3.4.2 eingehen, da sie für die vorliegende Arbeit geringfügig erweitert werden musste. Detailliertere Informationen über die Back-Annotation sind in [Gedikli, 2008] zu finden.

3.1.3. Schleifenanalyse

Wie bereits in Abschnitt 2.6 erwähnt benötigt die Analysesoftware aiT einige Informationen, wie z.B. Schleifeniterationsgrenzen und Rekursionsschranken, da diese im allgemeinen Fall nicht automatisch bestimmt werden können. Allerdings ist die in aiT integrierte Schleifenanalyse nur sehr eingeschränkt funktionsfähig, so daß in [Lokuciejewski et al., 2009a] eine in den WCC integrierte Schleifenanalyse umgesetzt wurde, die einen wesentlich größeren Funktionsumfang besitzt. Diese Schleifenanalyse ist nun fester Bestandteil des WCC und kann bei Bedarf eingesetzt werden, um die Iterationsgrenzen der im Programm enthaltenen Schleifen zu ermitteln. Für Rekursionsschranken sind jedoch nach wie vor Benutzerangaben unerläßlich. Allerdings ist Rekursion in eingebetteter Software ein eher seltenes Phänomen, so daß dies keine große Rolle spielt.

3.1.4. Flowfactmanager

Ursprünglich wurden die Schleifengrenzen und Rekursionsschranken über eine spezielle Eingabedatei, die AIS-Datei, an aiT übergeben. Wenn der Benutzer diese Datei allerdings selbst anlegt, benötigt er für die Identifikation der Schleifen deren Adresse im Binärprogramm (Diese Adresse ist auch in den CRL2-Dateien codiert, die die Konverter erzeugen). Das ist einerseits ein sehr umständliches Vorgehen und andererseits kann die Adresse der Schleife sich durch verschiedene Optimierungen oder Codetransformationen ändern, so daß es auch keine stabile Eingabe darstellen würde. Um diese Probleme zu vermeiden, wurde der Flowfactmanager in der Diplomarbeit von Daniel Schulte [Schulte, 2007] erstellt, mit dessen Hilfe die benötigten Benutzerangaben direkt im Quellcode spezifiziert werden können. Dabei können Schleifengrenzen (***Loopbounds***) annotiert werden, die die Iterationszahl einer Schleife beschränken (siehe Abbildung 3.2(a)) und Flußbeschränkungen (***Flowrestrictions***), die das Verhältnis zwischen den Ausführungshäufigkeiten zweier Blöcke global beschränken (siehe Abbildung 3.2(b)). Die annotierten Flowfacts werden vom Flowfactmanager automatisch bis zur CRL2-Datei weitergereicht. Außerdem bietet der Flowfactmanager Methoden an, um die Flowfacts nach erfolgten Codetransformationen (z.B. nach dem Kopieren einer Schleife) konsistent zu halten.



```
void foo( void ) {
  int i;

  _Pragma( „loopbound min 5 max 5“ )
  for ( i = 0; i < 5; i++ ) {
    _Pragma( „loopbound min 1 max 5“ )
    do {
      work();
    } while( condition() );
  }
}
```

(a) Beispiel: Schleifengrenzen

```
int foo( void ) {
  _Pragma( „marker call“ )
  int result = bar( 15 );

  _Pragma( „flowrestriction 1*bar
                    <= 15*call“ )
  return result;
}

int bar( int i ) {
  if ( i > 0 ) {
    return bar( i - 1 );
  } else {
    return 1;
  }
}
```

(b) Beispiel: Flußbeschränkungen

Abbildung 3.2.: Beispiele für die Annotation von Flowfacts im Quellcode

3.2. Die TriCore-Plattform

Die Zielplattform des WCC ist der Infineon TriCore TC1796, der Code ist ebenfalls kompatibel zum TC1797. Bei beiden handelt es sich um Prozessoren, die von Infineon als eingebettete Echtzeitsysteme konzipiert wurden und für die Verwendung im Automotive-Bereich beworben werden. Der TC1796 [Infineon, 2007] weist sowohl Elemente einer RISC CPU als auch Elemente eines DSPs auf. Ähnlich wie RISC Systeme besitzt er 3 Pipelines, von denen jeweils eine für arithmetische Befehle, Speicherzugriffe bzw. schnelle Schleifenbearbeitung zuständig ist, 32 Mehrzweckregister und eine Harvard-Architektur. Andererseits besitzt er DSP-typische Eigenschaften wie z.B. eine Multiply-Accumulate Unit, Integerarithmetik mit Sättigung, Packed Operations und einen gemischten 32/16-Bit Befehssatz. Die meisten Befehle (bis auf z.B. Gleitkommabefehle, für die eine FPU zur Verfügung steht) werden in einem Zyklus abgearbeitet, bei einer Taktfrequenz von 180Mhz. Weitere Merkmale sind ein Scratchpad-Speicher der als deterministischer, benutzergesteuerter Cache fungieren kann, ein Context Save Buffer, der bei Funktionsaufrufen automatisch Register 0-15 der vorhandenen 32 Mehrzweckregister sichert. Diese Register werden als unterer Kontext bezeichnet. Durch diese hardwareseitige Unterstützung können Funktionsaufrufe wesentlich effizienter erfolgen, da das Sichern der Register durch einzelne `MOV`-Befehle dann entfällt. Außerdem ist im Befehlssatz ein `LOOP`-Befehl enthalten, mit dem Schleifen in bis zu 2 Schachtelungsstufen ohne weiteren Overhead abgearbeitet werden können.

Da die Algorithmen aus der vorliegenden Arbeit hauptsächlich auf der ICD-C IR Ebene arbeiten und daher unabhängig von der verwendeten Hardware verwendbar sind, sei hier für weitere Details auf die Dokumentation von Infineon verwiesen ([Infineon, 2007]).

3.3. Herausforderungen bei der WCET-Optimierung

Bereits bei der Optimierung eines Programms für die ACET stößt man auf klare Grenzen. So ist es z.B. im allgemeinen unentscheidbar, ob eine Optimierung die Laufzeit eines Programmes verringern kann, ebenso wie es unentscheidbar ist, alle Stellen in einem Programm zu finden, an denen die Optimierung anwendbar ist ([Muchnick, 1997, S.319f]). Bei der Optimierung für die WCET fallen jedoch im Vergleich dazu noch einige andere Schwierigkeiten an.

```
int i,j;
int array[10][10];
for ( i = 0; i < 10; i++ ) {
  for ( j = 0; j < 10; j++ ) {
    a[i][j] = foo(i,j);
    if (a[i][j] > LIMIT) {
      work_special();
    }
  }
}
```

(a) Originaler Code

```
int i,j, temp;
int array[10][10];
for ( i = 0; i < 10; i++ ) {
  for ( j = 0; j < 10; j++ ) {
    temp = a[i][j] = foo(i,j);
    if (temp > LIMIT) {
      work_special();
    }
  }
}
```

(b) Optimierter Code

Abbildung 3.3.: Änderung der Basisblock-WCET durch Optimierungen

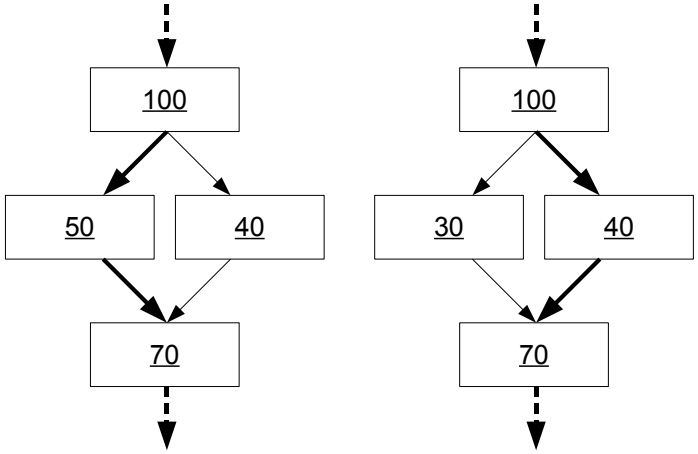
3.3.1. Verfügbarkeit und Aktualisierung von WCET-Informationen

Bei ACET-Optimierungen ist es leicht, einen Pfad zu bestimmen, den man optimieren möchte. Das gilt auch für globale Optimierungen, denn solange irgendein ausführbarer Pfad positiv von der Änderung betroffen ist (und nach Möglichkeit kein anderer Pfad negativ) ist die Auswirkung potentiell positiv für die ACET. Es besteht auch die Möglichkeit, über Probeläufe des Programms herauszufinden, welche Pfade besonders häufig ausgeführt werden (***Profiling***) und diese Pfade gezielt zu optimieren. Im Gegensatz dazu ist es wie in Kapitel 2 bereits erwähnt im höchsten Maße nichttrivial, den präzisen WCEP eines Programms zu bestimmen. Dieses Problem ist im WCC durch die Integration von aiT bereits gelöst, allerdings kann die Laufzeit einer einzelnen Analyse für komplexere Programme mit mehreren tausend Zeilen Quellcode bereits im Stundenbereich liegen.

Dazu kommt, daß die WCET-Informationen bei Änderungen am Inhalt der Basisblöcke durch einzelne Optimierungen aktualisiert werden müssen, um eine neue, aktuelle WCET des Gesamtprogramms zu erhalten. Wenn z.B., wie in Abbildung 3.3(a) im grau markierten Basisblock, die Neuberechnung eines gemeinsamen Ausdrucks durch eine temporäre Variable umgangen wird (3.3(b)), muß man entweder eine völlig neue WCET-Analyse mit aiT durchführen, die auch die WCET der Basisblöcke neu berechnen würde, oder versuchen, die Änderung der Basisblock-WCET näherungsweise abzuschätzen, falls die aiT-Analyse zu viel Zeit in Anspruch nehmen würde. Durch die Optimierungen können sich jedoch nicht nur die WCETs einzelner Basisblöcke ändern, sondern es kann zu ***Pfadwechseln*** kommen, die im nächsten Abschnitt genauer erläutert werden.

3.3.2. Pfadwechsel

Ein Problem, das bei der ACET-Optimierung in dieser Form überhaupt nicht auftritt, ist das Problem des Pfadwechsels. Als Beispiel sei eine Situation wie in dem Kontrollflußgraphen aus Abbildung 3.4(a) gegeben. Die Basisblöcke sind mit ihren WCETs beschriftet und der WCEP ist in schwarz markiert (im folgenden werden wir Basisblock-WCETs immer unterstrichen darstellen um sie von den an anderen Stellen ebenfalls angezeigten Kanten-WCECs zu unterscheiden). Durch eine Optimierung wird die WCET des linken Basisblocks um 20 Zyklen verringert, so daß der neue längste Pfad nun durch den rechten Basisblock verläuft (3.4(b)). Da nun auch die WCET des gesamten in Abbildung 3.4 sichtbaren Bereichs um 10 Zyklen gesunken ist kann sich dieser Effekt auf der nächst höheren Verzweigungsebene oder auf Funktionsebene weiterverbreiten. So wäre es zum Beispiel denkbar, daß die Funktion, deren WCET nun um 10 Zyklen verkürzt wurde, nach der Optimierung überhaupt nicht mehr auf dem WCEP liegt, weil sie innerhalb einer Verzweigung in einer anderen Funktion aufgerufen wurde. Hier ist also in jedem Fall eine globale Neuberechnung



(a) Vor dem Pfadwechsel (b) Nach dem Pfadwechsel

Abbildung 3.4.: Beispiel für einen Pfadwechsel

der WCET nötig um sicherzugehen, daß man noch auf dem WCEP arbeitet, da Optimierungen auf anderen Pfaden nicht zur WCET-Minimierung beitragen. Bei ACET-Optimierungen hat man dieses Problem nicht, weil die zu optimierenden Blöcke nur aufgrund ihrer Ausführungshäufigkeit ausgewählt werden und diese Eigenschaft durch rein lokale Optimierungen nicht zerstört werden kann.

Eine andere mögliche Lösung für das Problem der Pfadwechsel ist der in Abschnitt 3.1.2 bereits erwähnte Invariant Path, aber dieses Konzept ist im Zusammenhang mit den Superblockoptimierungen nicht direkt anwendbar. Auf die Gründe werden wir in Abschnitt 5.1 näher eingehen.

3.3.3. Einfluß lokaler Änderungen auf die globale WCET

Die Vorhersage lokaler Änderungen auf die WCET eines Programms, die wir, wie in 3.3.1 erläutert, benötigen werden, ist leider vielfach nur sehr schwer umsetzbar. So können z.B. durch das Verschmelzen zweier Basisblöcke möglicherweise Zyklen eingespart werden, da die evtl. dazwischenliegende Sprunginstruktion eingespart wurde. Allerdings kann es durch Pipeline-Hazards, fehlschlagende Cachezugriffe oder ähnliche Low-Level Effekte dazu kommen, daß durch die Verschmelzung zusätzliche Zyklen benötigt werden. Um diese Effekte abzuschätzen, benötigt man Informationen über den Zustand der Hardware, die nur im globalen Zusammenhang berechenbar sind, d.h. man benötigt für eine exakte Vorhersage eine erneute Analyse, so wie aiT sie durchführt. Da eine einzelne Analyse für größere Benchmarks bis zu mehrere Stunden dauern kann, und wir im Verlauf der Optimierungen den WCEP sehr oft neu berechnen bzw. überprüfen müssen, ist die wiederholte Analyse mit aiT nicht in jedem Fall praktikabel. Daher müssen wir in solchen Fällen teilweise auf heuristische Lösungen zurückgreifen.

Zusätzlich erschwert wird die Vorhersage der WCET-Änderung dadurch, daß lokale Änderungen an einem Basisblock nicht-lokale WCET-Änderungen hervorrufen können. So ist es z.B. denkbar, daß eine Transformation des Codes in Block b_i eine Änderung der WCET eines anderen Blockes b_j bewirkt. Der Grund hierfür kann in einem veränderten Code-Layout auf Binärebene oder veränderten Hardwarezuständen an den Übergängen von und zu Block b_i liegen. Je mehr Optimierungs- und Transformationsstufen zwischen der Darstellung, auf der die Neuberechnung der WCET-Daten stattfindet, und der Binärdarstellung liegen, desto schwerer wird es, diese Effekte zu berücksichtigen.

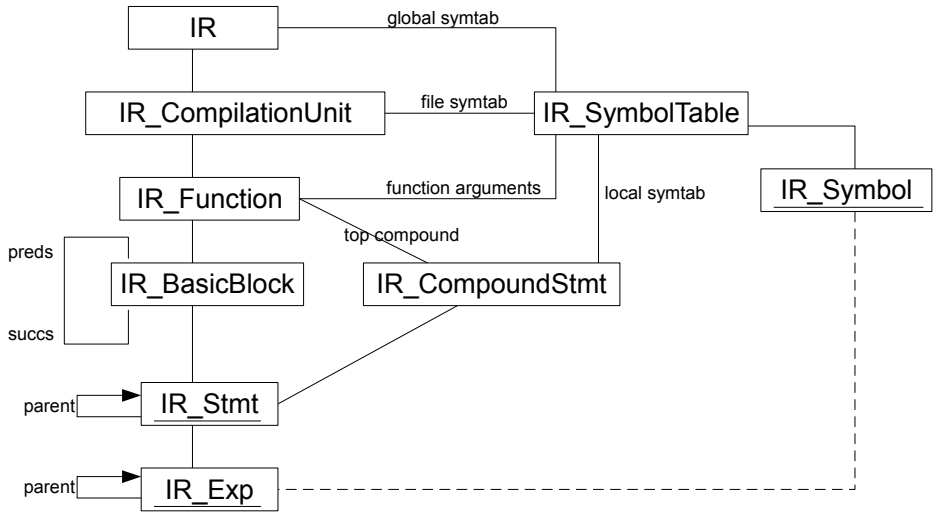


Abbildung 3.5.: Der Aufbau der ICD-C IR (abstrakte Klassen sind unterstrichen)

3.4. Arbeitsbereiche der Diplomarbeit

Die vorliegende Diplomarbeit stützt sich auf die vorhandenen Mechanismen des WCC, insbesondere jedoch auf die verwendete Zwischendarstellung ICD-C IR und auf die Back-Annotation. Daher sollen diese beiden Module im folgenden noch einmal ausführlicher erläutert werden.

3.4.1. ICD-C IR

Die ICD-C IR ist eine High-Level Darstellung des C Quellcodes, in der alle in C vorhandenen Kontroll- und Datenstrukturen direkt auf Objekte aus der in C++ geschriebenen ICD-C Bibliothek abgebildet werden. Daher ist eine Rückkonvertierung der IR in C Code ohne großen Aufwand möglich.

Aufbau der IR

Die ICD-C IR wird durch ein Objekt vom Typ `IR` dargestellt. Dieses Objekt verwaltet eine Liste von Übersetzungseinheiten vom Typ `IR_CompilationUnit`, die jeweils für eine einzelne Quellcodedatei stehen. Die Übersetzungseinheiten enthalten Funktionen (Typ `IR_Function`) und diese wiederum einzelne Anweisungen vom Typ `IR_Stmt`. Eine sequentielle Abfolge von Anweisungen innerhalb eines duch `{` und `}` begrenzten Blocks wird dabei zu einem `IR_CompoundStmt` zusammengefasst. Jede Funktion verweist daher nur auf das Compound Statement, daß ihren Funktionsrumpf repräsentiert.

In den Compound Statements, den Funktionen, den Übersetzungseinheiten und der IR werden die dort jeweils vorhandenen Symboldeklarationen in `IR_SymbolTable`s verwaltet. Diese Symbole können in Ausdrücken, die als `IR_Exp` Objekte dargestellt sind, benutzt werden. Die Ausdrücke sind jeweils mit dem Statement, in das sie eingehängt sind, verknüpft, so ist z.B. der Ausdruck `i<10` in `for( i=0; i<10; i++ )` mit dem Objekt der For-Anweisung verknüpft und der Ausdruck `w=k+t` wird in ein Objekt vom Typ `IR_ExpStmt` eingehüllt, damit er direkt als Anweisung dienen kann. Die gesamten Zusammenhänge sind noch einmal in Abbildung 3.5 dargestellt.

Die abstrakten Klassen `IR_Stmt` und `IR_Exp` sind jeweils in vielen Unterklassen spezialisiert worden, von denen jede für bestimmte C-Konstrukte steht. So gibt es neben den bereits vorgestellten

Unterklassen `IR_CompoundStmt` und `IR_ExpStmt` z.B. noch die Klassen:

- `IR_AsmStmt` für Inline-Assembler (vom WCC allerdings momentan nicht unterstützt)
- `IR_JumpStmt` (abstrakt) für Sprungbefehle, mit den Spezialisierungen
  - `IR_GotoStmt`,
  - `IR_ReturnStmt`,
  - `IR_BreakStmt` und
  - `IR_ContinueStmt`
- `IR_LoopStmt` (abstrakt) für Schleifenkonstrukte, mit den Spezialisierungen
  - `IR_ForStmt`,
  - `IR_WhileStmt` und
  - `IR_DoWhileStmt`
- `IR_TargetedStmt` (abstrakt) für Sprungmarken, mit den Spezialisierungen
  - `IR_CaseStmt`,
  - `IR_DefaultStmt` und
  - `IR_LabelStmt`
- `IR_SelectionStmt` (abstrakt) für Verzweigungskonstrukte, mit den Spezialisierungen
  - `IR_IfStmt`,
  - `IR_IfElseStmt` und
  - `IR_SwitchStmt`

Die Objekte des Typs `IR_Exp` gliedern sich ebenfalls in verschiedene Unterklassen, nämlich:

- `IR_AssignExp` für alle Zuweisungsoperatoren wie z.B. `=`, `+=` und `-=`
- `IR_BinaryExp` für alle binären Operatoren wie z.B. `+`, `-`, `*` und `/`
- `IR_CallExp` für Funktionsaufrufe
- `IR_ComponentAccessExp` für Zugriffe auf Structs und Unions, also für die `.` und `->` Operatoren
- `IR_ConstExp` für im Programm vermerkte Konstanten (Spezialisiert sich in `IR_IntConstExp`, `IR_FloatConstExp`, ... je nach Datentyp der Konstante)
- `IR_IndexExp` für Index-Zugriffe auf Arrays (`a[i]`)
- `IR_SymbolExp` für Zugriffe auf deklarierte Symbole (z.B. `a` und `i` im obigen Beispiel zur Index Expression)
- `IR_UnaryExp` für alle unären Operatoren wie z.B. `++`, `-`, `!` und `sizeof`

ICD-C Analysen

Wie man in Abbbildung 3.5 erkennen kann, gibt es noch einen zweiten Weg, auf die Anweisungen in einer Funktion zuzugreifen, nämlich über die Basisblockdarstellung. Eine Funktion wird dabei in ihre Basisblöcke zerlegt und diese können einzeln iteriert werden. Zu jedem Basisblock lassen sich dabei die Anweisungen des Blocks und die Nachfolger und Vorgänger im Kontrollflußgraphen abfragen. Die Basisblockdarstellung wird von der ICD-C bei Bedarf neu berechnet, insbesondere werden die vorhandenen Basisblockobjekte nach jeder kontrollflußrelevanten Änderung vollständig zerstört und bei der nächsten Anfrage neu aufgebaut. Die Basisblockdefinition der ICD-C entspricht im wesentlichen der aus Sektion 2.1, allerdings mit zwei Abweichungen: Funktionsaufrufe unterbrechen den aktuellen Basisblock *nicht*, und Operatoren mit implizitem Kontrollfluß, also `&&`, `||`und `?:`, tun dies ebenfalls *nicht*. Dies ist insbesondere bei der Zuordnung von IR-Blöcken zu LLIR-Blöcken ein Hindernis.

Als weitere Analyse stellt die ICD-C Def/Use Chains zur Verfügung. Diese ermöglichen es, an jeder `IR_SymbolExp` nachzufragen, wo sich die nächste Benutzung oder die nächste Definition des referenzierten Symbols befindet. Diese Analyse ist allerdings weder pointer-sensitiv noch interprozedural, d.h. falls sie auf Pointer oder Funktionsaufrufe stößt, muß sie die Def/Use Analyse abbrechen. Für diesen Fall geben die Abfragen der nächsten Benutzungen/Definitionen `NULL`-Pointer zurück. Da die Analyse außerdem fest in die einzelnen ICD-C Klassen integriert ist, ist eine Erweiterung um diese Funktionalität nicht möglich. Deswegen werden wir uns in Abschnitt 7.1 mit der Neuentwicklung einer pointer- und funktionssensitiven Lebendigkeitsanalse beschäftigen.

Die ICD-C IR bietet auch eine Schleifenanalyse, die allerdings nur einfache Schleifen mit festen Grenzen analysieren kann. Außerdem gibt es Iteratoren, die über ein gegebenes Statement und alle seine Unter-Statements iterieren, und Iteratoren, die über alle Ausdrücke iterieren, die direkt oder indirekt über ihren Elternausdruck in ein gegebenes Statement eingebunden sind.

ICD-C Optimierungen

Die ICD-C bietet bereits vielfältige klassische ACET-Optimierungen an. Diese sollen hier kurz erwähnt werden, da wir sie teilweise als Unterstützung für die Superblockbildung einsetzen werden. Teilweise sind diese Optimierungen fest in die ICD-C integriert und teilweise handelt es sich um Eigenentwicklungen des WCC-Teams. Die Optimierungen können im WCC über 3 verschiedene Optimierungsstufen gesteuert werden, mit zunehmender Optimierungsstufe werden mehr Einzeloptimierungen dazu geschaltet, angefangen bei O0 (keine Optimierung) bis O3 (alle Optimierungen). Der Ablaufplan der Optimierungen ist in Abbildung 3.6 dargestellt.

- **Common Subexpression Elimination**
  Ausdrücke, die bereits berechnet wurden und an späterer Stelle erneut berechnet werden sollen, ohne daß ihre Quellvariablen in der Zwischenzeit überschrieben wurden, werden erkannt und durch das zwischengespeicherte Ergebnis der ersten Berechnung ersetzt, so daß eine wiederholte Berechnung vermieden wird. Dies funktioniert in der ICD-C Implementierung nur für Anweisungen innerhalb desselben Basisblocks.

- **Dead Code Elimination**
  Unerreichbare Anweisungen und Anweisungen ohne Wirkung werden entfernt (z.B. Anweisungen nach einem `return` oder ein `if` mit einer konstanten Bedingung).

- **Fold Constant Code**
  Setzt den Wert von konstanten Variablen an ihren Verwendungsorten direkt ein und berechnet auch Ausdrücke, deren Argumente alle konstant sind, schon zur Kompilierzeit. Dies

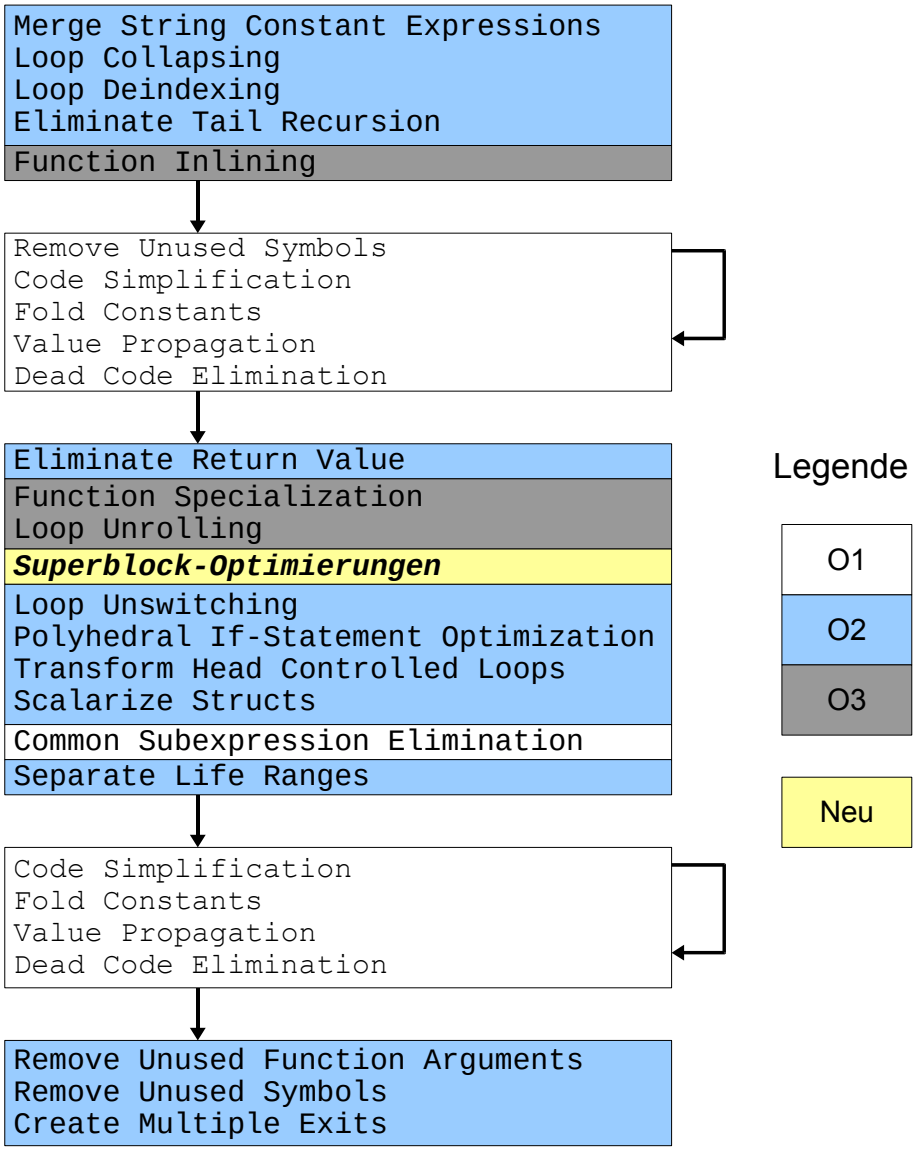


Abbildung 3.6.: Ablaufplan der im WCC verfügbaren High-Level-Optimierungen.

kann weitere Optimierungen ermöglichen, z.B. die Entfernung einer Schleife mit konstanter Bedingung `0` bzw. `false` (Dead Code Elimination).

- **Function Inlining**
  Kleinere Funktionen werden expandiert, indem ihr Funktionsrumpf an allen Stellen eingesetzt wird, an denen die Funktion aufgerufen wird.

- **Function Specialization**
  Falls Funktionsaufrufe mit konstanten Argumenten existieren, werden Kopien der Funktionen mit eingesetztem konstanten Paramter erzeugt und statt der Originalfunktion aufgerufen.

- **Loop Collapsing**
  Für verschachtelte Schleifen, die über ein mehrdimensionales Array iterieren, wird eine einzige neue Schleife generiert, die das Array in derselben Weise durchläuft, es aber dabei als eindimensionales Array behandelt.

- **Loop Deindexing**
  Arrayzugriffe innerhalb von Schleifen, bei denen der Index pro Iteration einer konstanten Veränderung unterliegt, werden durch einen Pointer ersetzt, der pro Iteration entsprechend in- oder dekrementiert wird.

- **Loop Unrolling**
  Schleifen mit bekannter Ausführungszahl werden um einen Faktor n abgerollt, so daß n-fach weniger Sprünge zum Schleifenkopf nötig sind. Dafür wird der Schleifenrumpf $n-1$ mal kopiert und die Aktualisierung der Schleifenvariable (bei `for` Schleifen) muß angepasst werden. Für Schleifen mit unbekannten Ausführungszahlen ist die Optimierung ebenfalls möglich, dort allerdings weit weniger effizient.

- **Loop Unswitching**
  Verzweigungen innerhalb von Schleifen, die unabhängig von den Schleifenvariablen sind, werden aus der Schleife hinausgezogen und passende Kopien der Schleife werden in den einzelnen Verzweigungen platziert.

- **Polyhedral If-Statement Optimization**
  Mithilfe von mathematischen Modellen wird versucht, if-Bedingungen zu finden, die redundant sind, d.h. deren Wert zur Laufzeit konstant ist. Diese können dann eliminiert werden. Besonders interessant ist auch diese Optimierung innerhalb von Schleifen, da dort mehr Laufzeit verbracht wird.

- **Tail Recursion Elimination**
  Direkt rekursive Funktionen, die sich selbst am Ende ihres Rumpfes neu aufrufen, werden zu einer nicht-rekursiven Funktion transformiert, indem der rekursive Aufruf durch einen Sprung zum Anfang der Funktion ersetzt wird.

- **Transform Head Controlled Loops**
  Kopfgesteuerte `for`- und `while`-Schleifen werden in fußgesteuerte `do-while`-Schleifen transformiert. Die do-while-Schleifen haben dabei den Vorteil, daß sie bei gleicher Iterationszahl einen Sprung weniger benötigen als ihre kopfgesteuerten Varianten. Die Optimierung ist auch für Schleifen möglich, deren Rumpf nicht immer mindestens einmal ausgeführt wird, allerdings ist dann ein `if` nötig, das die neue Schleife einschließt, so daß der Optimierungseffekt durch den Rücksprung aus dem `if` (je nach Code Layout) verschwinden kann.

- **Value Propagation**
  Setzt nach einer Zuweisung eines konstanten Werts an eine (nicht konstante) Variable diesen

Wert bei den nächsten Verwendungen der Variablen direkt ein, falls sichergestellt werden kann, daß die Variable in der Zwischenzeit nicht erneut überschrieben wurde (z.B. durch unbekannte Pointerzugriffe).

- **Hilfsoptimierungen**
  Remove Unused Symbols entfernt nicht gebrauchte Symbole und Funktionen aus dem Quellcode, **Remove Unused Function Arguments** macht dasselbe mit nicht benutzten Funktionsparametern. **Merge String Constant Expressions** fasst inhaltlich gleiche Stringkonstanten zu einer einzelnen Konstanten zusammen, so daß Speicher gespart wird. **Code Simplification** vereinfacht Ausdrücke mit Hilfe von Ersetzungsregeln, z.B. wird `&*a` zu `a` vereinfacht.

Die neu implementierten Superblock-Optimierungen werden in der Mitte der Optimierungskette aufgerufen, damit sie nicht auf völlig unoptimierten Code stoßen. Insbesondere ist es wichtig, daß die Dead Code Elimination, das Function Inlining und das Loop Unrolling vor den Superblockoptimierungen aufgerufen werden, da diese Optimierungen einerseits unnützen Code entfernen können, der in der Superblockbildung evtl sonst unnötiges Codewachstum verursachen würde (z.B. `if`-Statements mit leeren Verzweigungen), und da sie andererseits größere Stück zusammenhängenden Codes generieren können (Inlining/Unrolling), auf dem die Superblockbildung arbeiten kann. Da alle wichtigen elementaren Optimierungen auch nach der Superblockbildung noch einmal aufgerufen werden, ist hier auch sichergestellt, daß es genügend High-Level-Optimierungen gibt, die potentiell von den Transformationen, die die Superblockbildung vornimmt, profitieren können.

Erweiterbarkeit durch Benutzerannotationen

Da die ICD-C ein abgeschlossenes und kommerzielles Produkt ist, kann man die verwendeten Datenstruturen nicht direkt verändern. Da es allerdings häufig vorkommt, daß der Benutzer zusätzliche Informationen (z.B. WCET-Informationen) an Elemente der IR anhängen möchte, wurde für diesen Fall ein Erweiterungsmechanismus geschaffen, der genau dies erlaubt. Alle Klassen in der IR leiten sich vom Typ `IR_PersistentObject` ab. Dieser Typ stellt Methoden zur Verwaltung von angehängten Daten zur Verfügung, nämlich

- `void setUserData( const char *key, IR_PersistentObject *data )`
- `IR_PersistentObject *getUserData( ITEM t key ) const`
- `void removeUserData( ITEM t key )`

Über diese Methoden kann der Benutzer beliebige Objekte vom Typ `IR_PersistentObject` anhängen. Man kann auch ein einzelnes Objekt an mehrere IR-Objekte geichzeitig anheften. Da die gesamte IR und auch die anzuhängenden Objekte als dynamische Speicherobjekte allokiert sind, stellt die ICD-C auch eine Klasse `IR_AttachableObject` bereit, deren Objekte sich automatisch selbst löschen, wenn sie merken, daß sie an kein IR Objekt mehr angeheftet sind.

3.4.2. Back-Annotation

Das Konzept der Back-Annotation wurde in Abschnitt 3.1.2 bereits erläutert, allerdings soll hier noch einmal auf die Aspekte eingegangen werden, die für die folgenden Kapitel wichtig sind. Außerdem werden Aspekte besprochen, die für die vorliegende Arbeit in der Back-Annotation ergänzt werden mussten.

Zur Speicherung der annotierten Daten benutzt die Back-Annotation den Erweiterungsmechanismus der ICD-C, der im letzten Abschnitt (3.4.1) beschrieben wurde. Es gibt dabei 3 Annotationsklassen, die jeweils von `IR_AttachableObject` abgeleitet sind:

- `IR_WCETObject`: Die wichtigste Klasse, enthält die von aiT berechneten Werte, wie in 3.1.1 angegeben (mit Ausnahme der Kanten-WCECs).
- `IR_CodeSizeObject`: Enthält die Größe des Binärcodes zum annotierten Block in Bytes
- `IR_SpillCodeObject`: Enthält die Anzahl der Register-Spill-Instruktionen im annotierten Block

Die Back-Annotation wird durch den Codeselektor und die ausgeführten Optimierungen darüber informiert, welche LLIR-Blöcke zu welchen IR-Blöcken gehören. Der Codeselektor gibt hierbei die initiale Zuordnung vor und die Optimierungen verändern diese gegebenenfalls. Leider kann es hierbei zu Situationen kommen, in denen eine eindeutige Zuordnung nicht möglich ist:

1:0 Innerhalb der LLIR können Blöcke gelöscht werden, falls z.B. andere LLIR-Optimierungen die Instruktionen des Blocks gelöscht haben. In diesem Fall entsteht eine Situation, in der der IR-Block *keinen* passenden LLIR-Block hat. Dieser IR-Block wird dann als nicht auf dem WCEP liegend betrachtet.

n:1 Die ICD-C betrachtet z.B. die Schleifenbedingung einer `do-while`-Schleife immer als eigenen Basisblock und den Rumpf als einen anderen. Die LLIR hingegen behandelt ein Statement, das im Code direkt vor der `do-while`-Schleifenbedingung kommt, als Teil des Basisblocks der Schleifenbedingung. Hierdurch wird mehreren IR-Blöcken ein und derselbe LLIR-Block zugeordnet. Um diese Beziehung in der IR sichtbar zu machen, wird in diesem Fall nur *einer* der IR-Basisblöcke mit einem "echten" `IR_WCETObject` annotiert, der Rest bekommt ein `IR_WCETObject`, das eine Referenz auf den Block enthält, der die gemeinsamen WCET-Informationen hat. Diese Referenzen bei iterativen Codetransformationen konsistent zu halten, wird sich als nicht trivial erweisen.

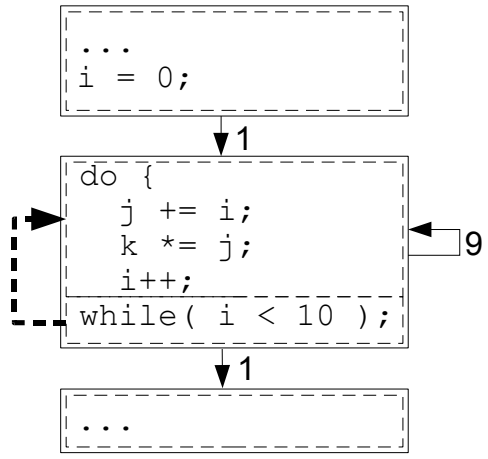
1:m Wie bereits erwähnt, führen Funktionsaufrufe und Ausdrücke mit implizitem Kontrollfluß (`&&`, `||`, `?:`) in der IR nicht zur Beendigung des Basisblocks, in der LLIR jedoch sehr wohl. So kann es also passieren, daß ein IR-Block mehreren LLIR-Blöcken zugeordnet wird. Die Annotationsdaten des IR-Blocks ergeben sich durch die geeignete Summierung der Daten der einzelnen LLIR-Blöcke: WCET-Werte, Codegröße und Spillcodezahl können aufaddiert werden, die Zugehörigkeit zum WCEP ist gegeben, falls einer der Blöcke auf dem WCEP liegt und Unausführbarkeit ist gegeben, falls alle LLIR-Blöcke unausführbar sind.

Neuerungen

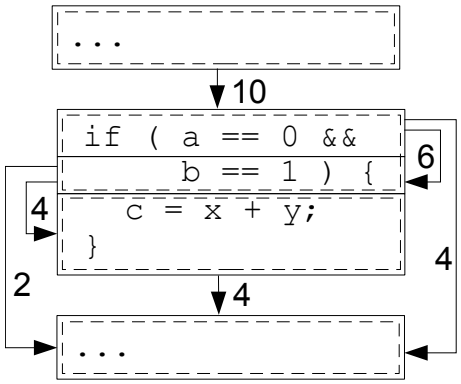
Für die Superblockoptimierungen werden zwei weitere Informationen an die IR-Blöcke annotiert, nämlich die ***Edge-WCEC***s also die Worst-Case-Ausführungshäufigkeiten der Kontrollflußkanten und eine ***Iteration WCET***, die die WCET pro Ausführung des Basisblocks angibt, im Gegensatz zur ebenfalls annotierten $WCET_{sum}$. Im folgenden wird skizziert, wie wir diese beiden Werte erhalten:

Edge-WCECs

Die Edge-WCECs sind in der LLIR Ebene als Wert direkt vorhanden. Dort wird jedem LLIR-Block b eine Abbildung von Nachfolgerblöcken auf die entsprechenden Kanten-WCECs zugeordnet. Zum Zeitpunkt der Übertragung der WCET-Daten sind die Abbildungen zwischen LLIR- und IR-Basisblöcken bereits fertig, d.h. die Auflösung der Nachfolgerblöcke ist ohne weiteres möglich. Im Falle einer 1:1 Zuordnung $b \rightarrow i$ eines LLIR- zu einem IR-Block i können wir die Werte daher direkt übernehmen, bei einer 1:0 Zuordnung wird der IR-Block und damit auch seine Edge-WCETs übersprungen, und bei einer n:1 Zuordnung wird immer nur der informationstragende Block annotiert, die Referenzblöcke müssen nicht extra annotiert werden, da sie Teil desselben LLIR-Basisblocks



(a) Bei einer n:1 Relation: Die do-while-Schleife ist auf LLIR Ebene ein einzelner Block, auf der IR jedoch in zwei Blöcke gespalten.



(b) Bei einer 1:m Relation: Der bedingte Ausdruck ist auf der IR Ebene ein einzelner Block, auf der LLIR jedoch in zwei Blöcke gespalten

Abbildung 3.7.: Beispiele für die Backannotation der Edge-WCECs.

sind. Abbildung 3.7(a) zeigt ein Beispiel hierzu. Die IR-Blöcke sind gestrichelt, die LLIR-Blöcke durchgezogen markiert, Referenzbeziehungen werden durch dicke, gestrichelte Pfeile dargestellt, und die auf LLIR vorhandenen Edge-WCECs stehen als Werte an den Kontrollflußkanten. In der Abbildung ist sichtbar, daß die Referenzblöcke (im Beispiel der do-while-Block) nicht gesondert behandelt werden müssen, da die informationstragenden Blöcke (im Beispiel der Schleifenrumpf) bereits alle verfügbaren Informationen verwalten.

Der einzige problematische Fall tritt auf, wenn eine m:1 Beziehung zwischen LLIR- und IR-Blöcken vorliegt (siehe Abbildung 3.7(b)). Wenn dieser Fall bei der Back-Annotation erkannt wird, müssen wir genau die LLIR-Kanten-WCECs zwischen den m Kanten, die zu einem IR-Block zusammengefasst wurden, ignorieren. Im Beispiel ist das die LLIR-Kante zwischen den einzelnen Teilen der if-Bedingung. Wir erkennen diese LLIR-Kanten daran, daß ihr LLIR-Zielblock (im Beispiel `b == 1`) ungleich dem LLIR-Startblock ist (im Beispiel `if ( a == 0 ... )`), aber wieder auf den IR-Block abgebildet wird, dessen Edge-WCECs wir bestimmen (im Beispiel `if ( a == 0 && b == 1 )`). Dieses Vorgehen ignoriert exakt diejenigen LLIR-Edge-WCECs zwischen zwei verschmolzenen LLIR-Basisblöcken, denn :

1. Wir nehmen an, wir ignorieren auf diese Weise eine LLIR-Edge-WCEC auf einer Kontrollflußkante $e = (b_i, b_j)$ die nicht zwischen zwei verschmolzenen LLIR-Blöcken b_i und b_j verläuft (ignorieren also zuviel). Da beide aber auf denselben IR-Block B abgebildet werden, muß $b_i = b_j$ gelten, was einen Widerspruch zur Annahme $b_i \neq b_j$ darstellt.

2. Wir nehmen an, wir ignorieren auf diese Weise eine LLIR-Edge-WCEC auf einer Kontrollflußkante $e = (b_i, b_j)$ zwischen zwei verschmolzenen LLIR-Blöcken b_i und b_j *nicht*. Die verschmolzenen Blöcke werden, wenn wir die Korrektheit der Abbildung LLIR-Blöcke $\rightarrow$ IR-Blöcke voraussetzen, beide auf einen gemeinsamen IR-Block B abgebildet. Wir können die Kante also nur dann nicht als Kante zwischen zwei verschmolzenen Blöcken erkennen, wenn $b_i = b_j$ gilt, was allerdings ein Widerspruch zu der Annahme ist, daß es sich um zwei verschmolzene LLIR-Blöcke handelt.

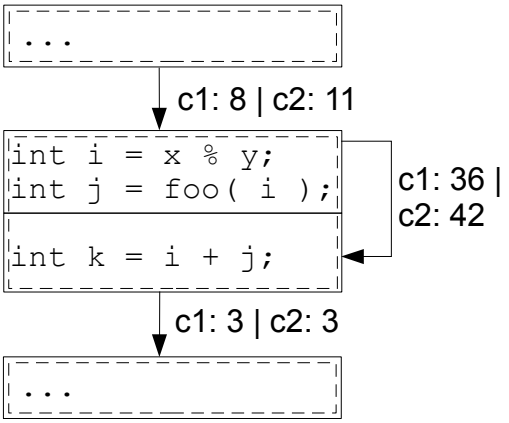


Abbildung 3.8.: Beispiel zur Berechnung der Iteration WCET in einer 1:m Relation.

Iteration-WCET

Die Iteration-WCET erhalten wir bei der Durchführung der Back-Annotation relativ einfach über die Auswertung von in der LLIR enthaltenen Kanteninformationen. Zu jeder Kante des Kontrollflußgraphen ist in der LLIR eine Kanten-WCET einsehbar, die angibt, wie hoch die WCET des Basisblocks für eine einmalige Ausführung ist, wenn der Block auf der gewählten Kante verlassen wird. Wir ermitteln also die Iteration-WCET des Knotens als Durchschnitt über alle Kanten-WCETs. Falls wir eine 1:m Situation haben und der Knoten bereits eine Iteration-WCET besitzt, können wir den ermittelten Durchschnitt einfach auf die bestehende Iteration-WCET aufaddieren, da es sich bei beiden Werten um Teile desselben IR-Blocks handelt. Ein Beispiel dazu findet sich in Abbildung 3.8. Die Kanten-WCETs sind dort pro Kontext (Annahme zweier Kontexte *c1* / *c2*) an den Kontrollflußkanten notiert und die LLIR- bzw. IR-Blöcke sind wieder wie im letzten Beispiel markiert. Der obere IR-Basisblock erhielte die Iteration-WCET $(8 + 11)/2 = 9$ (abgerundet) und dem mittleren Basisblock würde eine Iteration-WCET von $(36+42)/2+(3+3)/2 = 42$ zugewiesen. In den 1:0 und n:1 Konstellationen sind, wie bei den Edge-WCECs und aus denselben Gründen wie dort, keine weiteren Arbeiten nötig.

Persistenz der annotierten Informationen

In der ursprünglichen Back-Annotation wurden die Informationen direkt an die IR-Basisblöcke geheftet. Hierbei stellt sich das Problem, daß diese Block-Objekte nach jeder kontrollflußrelevanten Änderung an der Funktion komplett gelöscht und neu aufgebaut werden. Die Annotationen bemerken in diesem Fall, daß sie mit keinem Objekt mehr verknüpft sind und werden ebenfalls gelöscht. Wie wir in Kapitel 6 sehen werden, ist es bei der Superblockbildung nötig, iterativ auf einer Funktion zu arbeiten, d.h. wir optimieren erst einen Teil der Funktion und möchten danach immer noch auf die WCET-Daten zugreifen können. Dies war ursprünglich nicht möglich.

Als Lösung wurde eine Konverterklasse `AnnotationConverter` geschaffen, die die Annotationen bei Bedarf von den Basisblöcken entfernt und eine Kopie an das jeweils erste Statement jedes Basisblocks anhängt (ein Klassendiagramm zu den folgenden Erläuterungen ist in Abbildung 3.9 dargestellt). Die Statements, und damit auch ihre Annotationen, bleiben normalerweise im Verlauf der Optimierung erhalten. Falls eine Optimierung ein Statement verschiebt oder entfernt, ist diese Optimierung auch dafür verantwortlich, daß am Start-Statement jedes Basisblocks danach wieder konsistente Informationen stehen. Die Konvertierung der Annotationen wird dabei über zwei abstrakte Klassen durchgeführt: Alle vorhandenen Annotationen wurden so aufgerüstet, daß sie von den neuen Klassen `IR_BasicBlockAnnotation` und `IR_StatementAnnotation` erben.

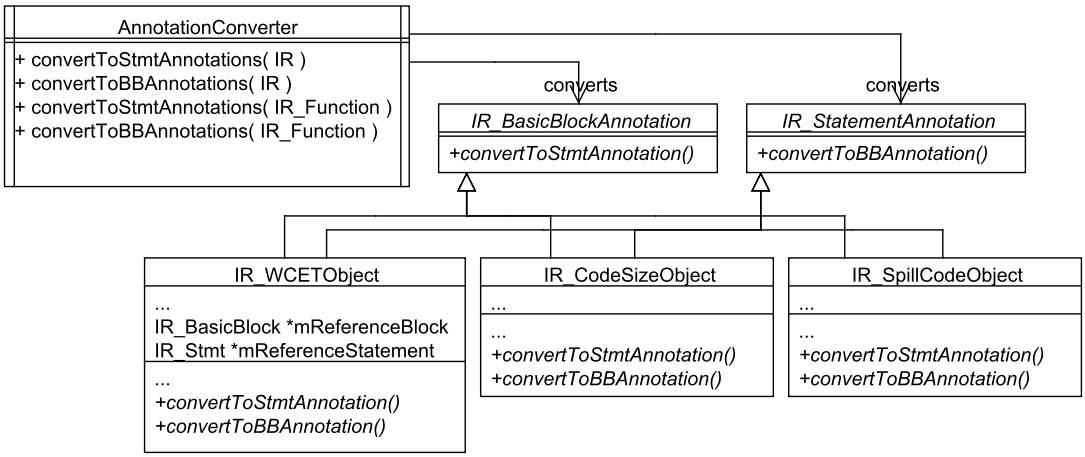


Abbildung 3.9.: Aufbau des Annotationskonverter-Framworks.

Diese Klassen stellen abstrakte Methoden zur Verfügung, mit deren Hilfe die jeweilige Basisblock-Annotation in eine Statement-Annotation konvertiert werden kann und umgekehrt. Die einzelnen Klassen wie z.B. `IR_WCETObject` implementierten dann die Konvertermethoden, wobei z.B. die in den WCET Objekten gespeicherten Referenzblöcke zu Referenzstatements konvertiert werden. Statt die Basisblock- und Statement-Version jeder Annotation in getrennten Klassen zu implementieren, wurden jeweils beide Versionen in nur einer Klasse implementiert, da die Ähnlichkeiten zwischen den beiden Klassen sonst sehr groß wären. Inhaltlich konvertiert werden in allen Fällen nur die Verknüpfungen zu anderen Basisblöcken bzw. Statements, wie z.B. die Referenzbeziehung. Dies bedeutet, daß z.B. Objekte vom Typ `IR_WCETObject` zur Laufzeit immer nur *eine* der beiden Funktionen erfüllen.

4. Superblöcke

In diesem Kapitel wird das in der Literatur bereits bekannte und ausführlich diskutierte Konzept der Superblöcke dargestellt. In Abschnitt 4.1 wird das ältere Konzept der Traces erläutert, Abschnitt 4.2 führt die Superblöcke ein und diskutiert ihre Umsetzung in verschiedenen Publikationen. In Abschnitt 4.4 werden abschließend die in der Literatur vorgestellten Optimierungen auf Superblöcken vorgestellt.

4.1. Traces als Vorläufer der Superblöcke

Bei einem ***Trace*** T handelt es sich um einen Pfad $(b_a, ..., b_e)$ durch den Kontrollflußgraphen, wobei hier üblicherweise der Low-Level-Kontrollflußgraph gemeint ist. b_a bezeichnen wir als ***Anfang*** des Traces, und b_e analog dazu als ***Ende*** des Traces. Traces wurden zum ersten mal von Fisher in [Fisher, 1981] erwähnt und für das Trace Scheduling benutzt. Hierbei wurden Low-Level-Anweisungen auf dem Trace verschoben (scheduled), so daß sich für den gewählten Pfad unter Beachtung der Hardwareabhängigkeiten eine nach Möglichkeit kürzere Ausführungszeit ergab. Dies setzt eine detaillierte Kenntnis und Modellierung der verwendeten Hardware voraus und soll hier nicht weiter vertieft werden. Der Kernpunkt ist, daß erstmals bestimmte Pfade gezielt optimiert werden sollten. Für die Auswahl (Selektion) der Traces wurden bei Fisher erwartete Ausführungshäufigkeiten der Basisblöcke benutzt, die als Eingabe für den Algorithmus fungieren. Diese wurden in späteren Arbeiten durch für reale Eingaben gemessene Ausführungshäufigkeiten von Basisblöcken ($w(b)$ für alle $b \in V$) und Kontrollflußkanten ($w(e)$ für alle $e \in E$) ersetzt. Aufbauend auf diesen erwarteten oder gemessenen Häufigkeiten wurden von Fisher und nachfolgend von Chang [Chang & Hwu, 1988] zwei Heuristiken zur Traceselektion präsentiert. Beide haben gemeinsam, daß die Erzeugung eines Traces T_i ausgehend von einem Startknoten b_{seed} durchgeführt wird. Dieser wird in beiden Arbeiten als derjenige Knoten mit der höchsten Ausführungsfrequenz $w(b_{seed})$ gewählt, der noch nicht Teil eines anderen Traces T_j ist. Von diesem Knoten aus wird der Trace $T_i = (b_a, ..., b_{seed}, ...b_e)$ abwechselnd am Anfang und am Ende verlängert. Die Menge aller bisher selektierten Traces bezeichnen wir im folgenden mit $Traces$. Die Heuristiken sind im folgenden für den Fall der Verlängerung am Ende des Traces bespielhaft dargestellt, die Verlängerung am Anfang funktioniert analog.

- **Selektion durch Knotengewichte**: Wähle b_{new} mit $(b_e, b_{new}) \in E$, $\forall T_j \in Traces : b_{new} \notin T_j$ und $w(b_{new}) = \max\{w(b_i) | (b_e, b_i) \in E\}$. Dies ist der einfachste Ansatz.
- **Selektion durch Kantengewichte**: Wähle b_{new} mit $e_{new} = (b_e, b_{new}) \in E$, $\forall T_j \in Traces : b_{new} \notin T_j$ und $w(e_{new}) = \max\{w(e) | e \in \delta^+(b_e)\}$. Die Idee ist hierbei, daß durch diese Art der Selektion Fälle vermieden werden, in denen zwei benachbarte Knoten zwar hohe Ausführungsfrequenzen haben, die Kante zwischen ihnen jedoch nur selten benutzt wird. Diese Kante würde in der ersten Heuristik gewählt, obwohl sie keinen häufig ausgeführten Pfad darstellt. So würde die Knotenselektion z.B. im Abbildung 4.1 den 'false' Pfad wählen, obwohl derjenige über 'true' hier der am häufigsten ausgeführte ist. Die Knoten- bzw. Kantengewichte sind in der Abbildung an ihren jeweiligen Knoten bzw. Kanten platziert, die Knotengewichte sind dabei wie auch schon in vorhergehenden Beispielen unterstrichen dargestellt.

Chang schlägt außerdem eine Erweiterung vor, bei der die Traceselektion abgebrochen wird, sobald ein Knoten erreicht wird, bei dem keine Kante mehr eine bestimmte, im vorhinein festgelegte Wahr-

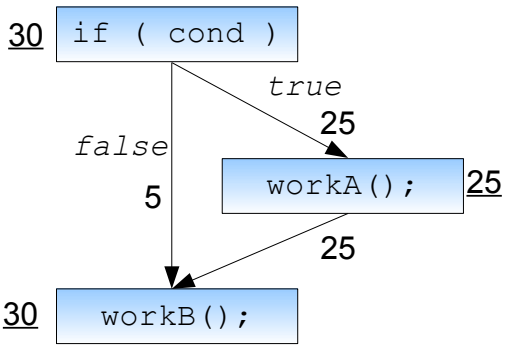


Abbildung 4.1.: Beispiel für Versagen der Selektion durch Knotengewichte.

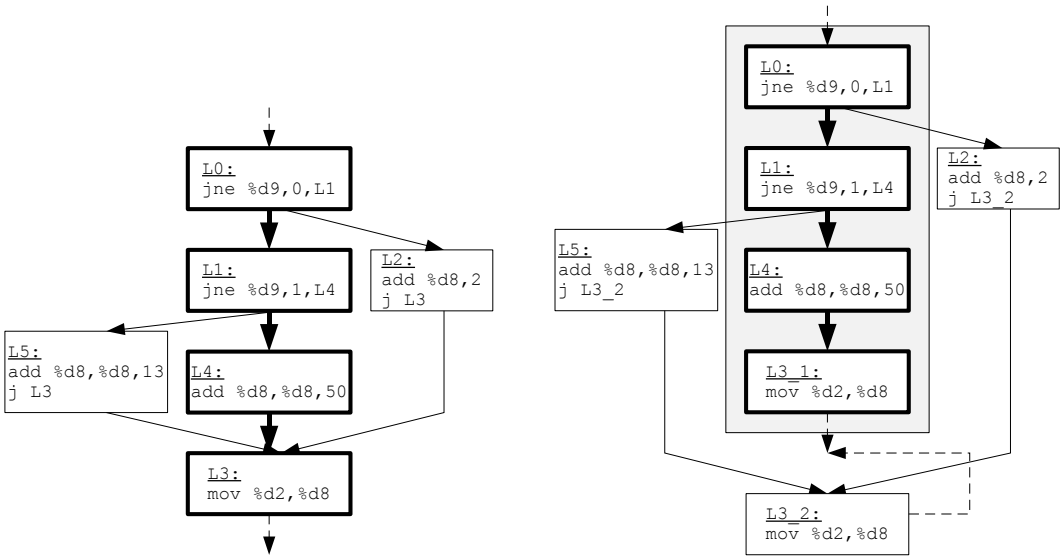
scheinlichkeit überschreitet. Die Wahrscheinlichkeit einer Kante $e = (b_i, b_j)$ ist dabei als $w(e)/w(b_j)$ bzw. $w(e)/w(b_i)$ definiert, je nachdem, ob nach vorne oder nach hinten verlängert wird. Dies soll die Traceselektion stoppen, sobald kein eindeutig bevorzugter Pfad mehr vorhanden ist.

Nach der Traceselektion führt Fisher auf dem Trace ein List-Scheduling durch, wobei nur die Abhängigkeiten zwischen Anweisungen auf dem Trace beachtet werden. Daher kann es vorkommen, daß dabei Anweisungen über Basisblockgrenzen hinaus verschoben werden, z.B. aus einer Verzweigung heraus (spekulative Ausführung) oder über einen Einsprungpunkt hinaus. Ein ***Einsprungpunkt*** ist dabei ein Block b_{in} auf dem Trace T, der außer seinem Vorgänger b_t mit $(b_t, b_{in}) \in T$ einen weiteren Vorgänger b_{off} mit $b_{off} \notin T$ hat. Um die Semantik des Programms zu erhalten, muss in solchen Fällen Kompensationscode erzeugt werden, nachdem die Anweisungen auf dem Trace verschoben wurden. Vor allem die Bewegungen über Einsprungpunkte hinweg führten zu einer aufwendigen Sonderbehandlung.

Nach dem List-Scheduling werden iterativ weitere Traces gesucht, wobei Fisher innere Schleifen zuerst bearbeitet und danach durch "Loop Representatives" ersetzt. Diese können dann im weiteren Verlauf wie eine elementare Anweisung behandelt werden. Dieses Konzept werden wir auch bei den High-Level-Superblöcken in Abschnitt 4.3 wieder aufgreifen.

4.2. Low-Level-Superblöcke

Um die an den Einsprungpunkte entstehenden Komplikationen zu vermeiden, entwickelte Chang [Chang et al., 1991] das Konzept des Superblocks. Ein ***Superblock*** ist ein Trace, der außer an seinem Startknoten keine Einsprungpunkte aufweist, formal also ein Trace $T = (b_{start}, ..., b_{end})$ für den $\forall b \in T \setminus \{b_{start}\} : \forall (b_i, b) \in E : b_i \in T$ gilt. In Abbildung 4.2(a) ist beispielsweise jeder Trace bzw. Pfad ein Superblock, außer den Pfaden, die Block L3 beinhalten. Chang beschränkt sich aber nicht darauf, die im Programm auf natürliche Weise vorhandenen Superblöcke zu nutzen, sondern transformiert beliebige schleifenfreie Traces, die nach den Kantengewichten selektiert werden, zu Superblöcken. Hierfür wird für jeden Einsprungpunkt b_{in} in einem Trace $T = (b_a, ..., b_{trc}, b_{in}, ..., b_e)$ eine exakte Kopie der Knoten $b_{in}, ..., b_e$ angelegt. Dieses Vorgehen bezeichnet man als ***Tail Duplication***, da das Trace-Ende dupliziert wird. Alle Einsprungkanten $e \in \{(b_{pred}, b_{in}) \in E | b_{pred} \neq b_{trc}\}$ werden dann zur erzeugten Kopie des Knotens b_{in} umgeleitet. Ein Beispiel für die Tail Duplication oder auch Superblockbildung ist in Abbildung 4.2 skizziert. Der gewählte Trace, der zu einem Superblock transformiert wird, ist dort fett markiert. Diese relativ einfache Superblockbildung ist auf der Low-Level-Ebene möglich, da durch die Verwendung von Sprungmarken der Kontrollfluß innerhalb des kopierten Abschnitts derselbe bleibt wie im Originalcode (bis auf einen evtl. vorhan-



(a) Das ursprüngliche Programm. (Gewählter Trace ist markiert)

(b) Programm nach der Low-Level Tail Duplication. (Superblock ist markiert)

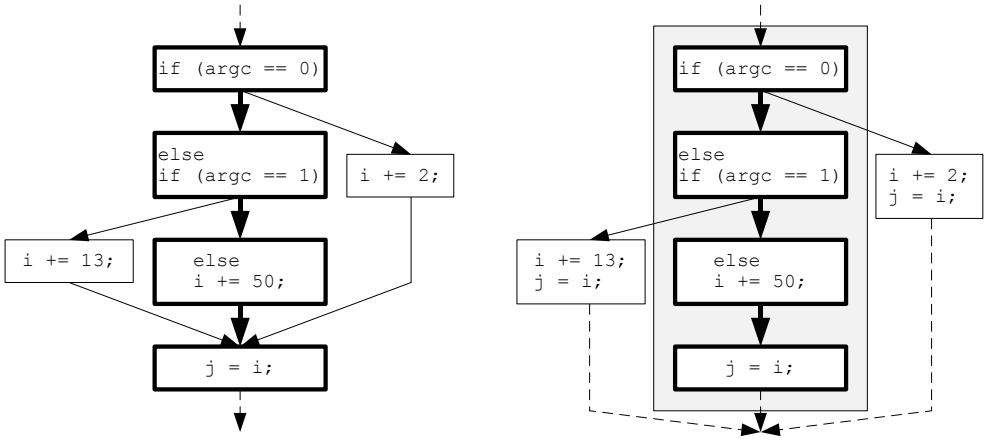
Abbildung 4.2.: Ein Beispiel für Low-Level Tail Duplication.

denen Fall-Through-Pfad am Traceende). Dies ist bei strukturiertem High-Level-Code nicht ohne weiteres gegeben.

4.3. High-Level-Superblöcke

Auf der High-Level-Ebene lassen sich Superblöcke ganz analog zu Abschnitt 4.2 definieren, so daß ein ***High-Level-Superblock*** ein einsprungfreier Pfad auf dem High-Level-Kontrollflußgraphen ist. Unsere Definition von High-Level-Superblöcken unterscheidet sich allerdings in einem Punkt von dieser einfachen Übertragung: Analog zum Vorgehen von Fisher lassen wir zu, daß ein Superblock $S = (b_0, b_1, ..., b_n)$ innere Schleifen beinhaltet, wobei wir eine Schleife als ***innere Schleife*** ansehen, wenn der Schleifenkopf ein Basisblock b_{head} ist mit $b_{head} = b_k$ für $k \in \{1, ...n\}$. Der Block b_{k+1} ist dann der nächste Block *nach* der inneren Schleife. Da wir, wie wir im nächsten Kapitel sehen werden, die Superblockbildung nur auf ***strukturierte Programme***, d.h. Programme ohne `gotos`, anwenden werden, ist die Erkennung der inneren Schleifen trivial. Da wir nur mit strukturiertem High-Level-Code arbeiten werden, entfällt auch die Problematik, daß durch die Code-Restrukturierungen Schleifen von den Low-Level-Analysen nicht mehr als solche erkannt werden können, wie z.B. in [Kidd & Hwu, 2006] berichtet. Die High-Level-Übertragung des Beispiels aus Abbildung 4.2 findet sich in Abbildung 4.3. Die einzelnen Traceblöcke sind hier und im folgenden als Kästen markiert, die die Instruktionen des Blocks enthalten. Die eingezeichneten Pfeile zeigen den Verlauf des Superblock bzw. Traces an. Man erkennt hier bereits einen Unterschied in der Erzeugung der Superblöcke: Das einmalige Kopieren des Trace-Endes reicht hier nicht in jedem Fall aus, da sich der Kontrollfluß im strukturierten High-Level-Programm nicht beliebig verändern lässt. Dieses Problem werden wir im nächsten Kapitel näher betrachten.

Da die High-Level-Superblockbildung kein Scheduling nach sich ziehen kann, da dies nur auf der Low-Level-Ebene sinnvoll durchführbar ist, stellt sich die Frage, welche Effekte wir uns davon versprechen. Die angestrebten Effekte sind unter anderem:



(a) Das ursprüngliche Programm. (Gewählter Trace ist markiert)

(b) Programm nach der High-Level Tail Duplication.

Abbildung 4.3.: Ein Beispiel für High-Level Tail Duplication.

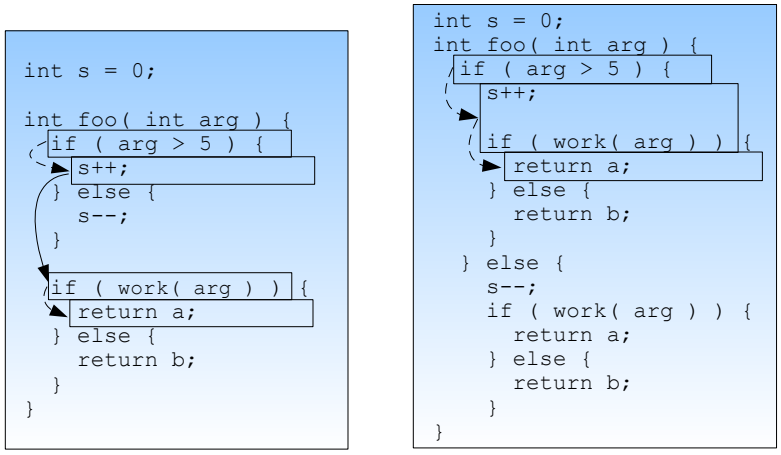
- **Erhöhtes Optimierungspotential**: Durch die Tail Duplication werden Datenabhängigkeiten aufgelöst. Wo zuvor Abhängigkeiten zu allen Vorgängerblöcken berücksichtigt werden mussten, müssen nach der Superblockbildung nur noch die Abhängigkeiten zum Superblockpfad selbst berücksichtigt werden. Hiervon profitieren Standard-Optimierungen wie z.B. Constant- und Value-Propagation, Constant Folding, Dead Code Elimination und Lokale Common Subexpression Elimination. Spezielle hierauf aufbauende Optimierungen werden in Abschnitt 4.4 vorgestellt.

- **Sprungvermeidung**: Wenn ein Superblock über eine Verzweigung hinweg bis zum Ende einer Schleife reicht, oder ein nachfolgendes `if`, `switch` oder eine kopfgesteuerte Schleife umfasst, dann entfällt der Rücksprung aus der ersten Verzweigung (siehe Beispiel in Abbildung 4.4, Sprünge sind durch durchgezogene Kanten, Fall-Through-Pfade durch gestrichelte Kanten gekennzeichnet).

- **Analysegenauigkeit**: Die Analysen von aiT und viele andere statische Analysen, wie z.B. die Schleifenanalyse und die Aliasanalyse des WCC funktionieren nach dem Prinzip der ***abstrakten Interpretation*** [Cousot & Cousot, 1977]. Dabei muß an Punkten, an denen der Kontrollfluß sich vereinigt, eine sichere Überabschätzung der Informationen auf den einzelnen Pfaden gebildet werden. Die Superblockbildung reduziert die Anzahl der Vereinigungspunkte und erhöht somit potentiell die Präzision der Analyse.

- **Vorbereitung für ein Scheduling**: Die auf der High-Level-Ebene erzeugten Superblöcke werden sich auch in der Codestruktur der LLIR widerspiegeln und daher dort ein basisblockübergreifendes Scheduling durch die gesenkte Zahl der Abhängigkeiten vereinfachen.

Die wichtigsten Punkte sind hierbei der erste und zweite, die in dieser Form auch als Zielsetzungen bei der Entwicklung anderer struktureller Codetransformationen wie z.B. dem Loop Unrolling oder dem Function Inlining dienen. Da auch diese bereits erfolgreich auf der High-Level-Ebene angewandt wurden, wird in dieser Arbeit zum ersten Mal die Übertragung der Superblockbildung auf die High-Level-Ebene untersucht.



(a) Das ursprüngliche Programm (Trace-Blöcke sind markiert).

(b) Programm nach der High-Level Tail Duplication. Der Rücksprung aus dem ersten `if`-Statement entfällt.

Abbildung 4.4.: Vermeidung von Rücksprüngen durch Superblockbildung.

4.4. Superblockoptimierungen

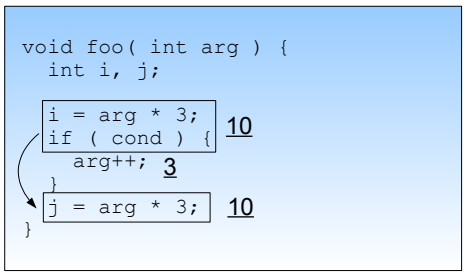
Zusätzlich zur Erzeugung der Superblöcke aus Traces wurden in der Literatur einige Optimierungen vorgestellt, die speziell die Optimierung der Superblöcke zum Ziel hatten. Die wichtigsten sollen hier kurz vorgestellt werden, da wir in Kapitel 7 zwei der Optimierungen als WCET-gesteuerte und -sensitive Optimierungen umsetzen werden. Wir bezeichnen eine Optimierung hierbei als ***WCET-gesteuert***, wenn sie ihre Entscheidungen mit Hilfe von WCET-Daten trifft, und wir bezeichnen sie als ***WCET-sensitiv***, wenn sie die im Programm vorliegenden WCET-Informationen in Verlauf der Optimierungen aktualisiert und konsistent hält

Common Subexpression Elimination

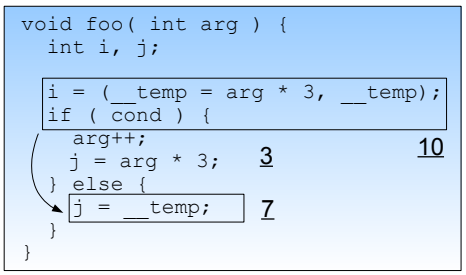
Die ***Superblock Common Subexpression Elimination*** (SB-CSE) hat zum Ziel, Berechnungen von Ausdrücken, die auf dem Superblock mehrfach ausgewertet werden sollen, zu eliminieren. Der Wert des Ausdrucks bei der ersten Berechnung wird hierbei in einer temporären Variable ablegt und die folgenden Berechnungen lesen nur noch diese Variable, statt den Wert des Ausdrucks neu zu ermitteln, zumindest solange, wie keine der Variablen, die der ersetzte Ausdruck liest, überschrieben wurde. Die Optimierung ist innerhalb von Basisblöcken, innerhalb von Superblöcken und auch global anwendbar, wobei die globale Form in ihrer modernsten Umsetzung eine spezielle Darstellung des Codes, die Static Single Assignment Form [Muchnick, 1997], benötigt, die im WCC bisher nicht vorhanden ist. Die lokale CSE ist allerdings, wie in Abschnitt 3.4.1 erwähnt, im WCC bereits verfügbar. Daher werden wir uns in dieser Arbeit auf die Superblock-CSE beschränken. Ein Beispiel für die Vorteile, die die Superblock-basierte CSE gegenüber der Basisblock-basierten CSE bietet ist in Abbildung 4.5 dargestellt. Beispielwerte für die Ausführungshäufigkeiten der Basisblöcke sind unterstrichen dargestellt. In Teil 4.5(a) ist die Ersetzung des Ausdrucks `arg * 3` noch nicht möglich, nach der Superblockbildung ist sie es jedoch.

Dead Code Elimination / Operation Migration

Die klassische Dead Code Elimination entfernt Code aus dem Programm, der nie ausgeführt werden kann, also tot ist, oder der keinen Effekt hat. Bei der ***Superblock Dead Code Elimination*** (SB-DCE) wird hingegen Code, der im Superblock selbst keinen Effekt hat, aus dem Superblock heraus

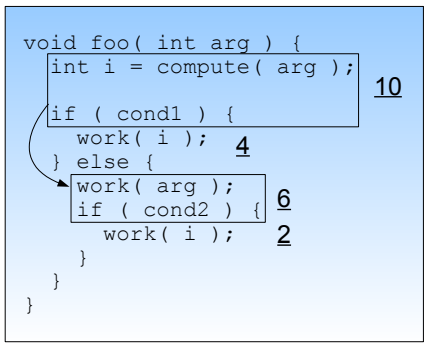


(a) Das ursprüngliche Programm.

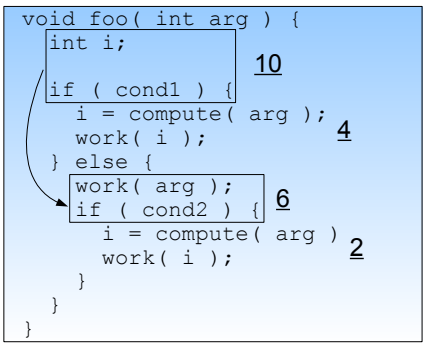


(b) Programm nach der High-Level Tail Duplication und Superblock-CSE

Abbildung 4.5.: Ein Beispiel für die Superblock-CSE.



(a) Das ursprüngliche Programm.



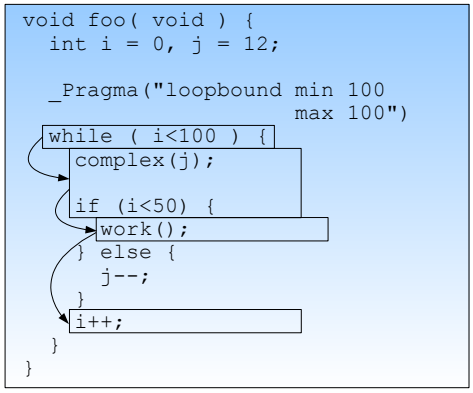
(b) Programm nach der Superblock Dead Code Elimination

Abbildung 4.6.: Ein Beispiel für die Superblock-DCE.

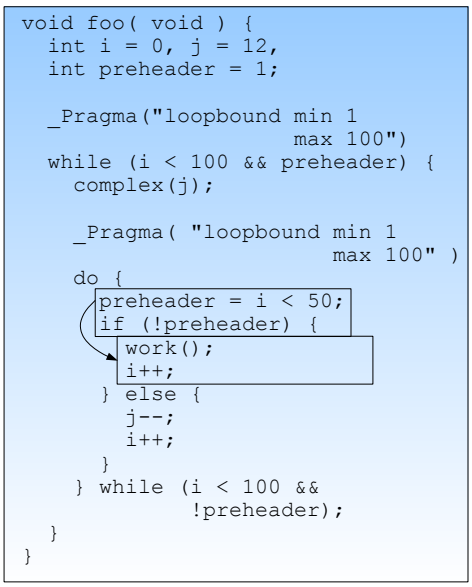
geschoben. Die Absicht hierbei ist, den Superblock zu verkürzen und somit das Optimierungskriterium (ACET/WCET), nach dem er gewählt wurde, zu verbessern, indem die Ausführungshäufigkeit des verschobenen Codes gesenkt oder der WCET-Pfad verkürzt wird. Ein anschauliches Beispiel hierfür findet sich in Abbildung 4.6 (auch dort sind wieder beispielhafte Ausführungshäufigkeiten der Basisblöcke unterstrichen dargestellt). Solcher Code kann z.B. durch vorangehende Optimierungen oder durch die Verwendung von Makros im Originalprogramm entstehen.

Loop Invariant Code Removal

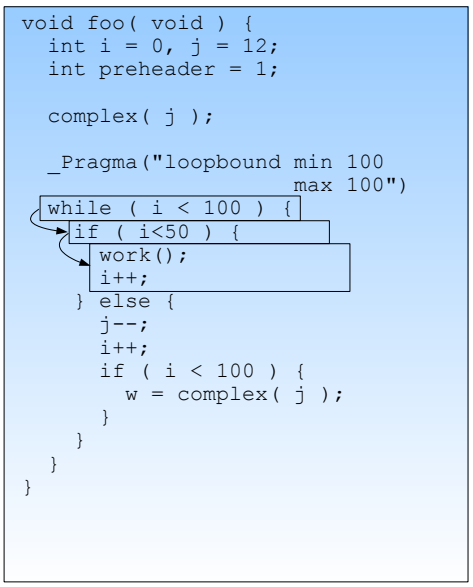
Auch hier handelt es sich wieder um eine Spezialisierung einer bekannten ACET-Optimierung. Beim Loop Invariant Code Removal, manchmal auch als Loop Invariant Code Motion bezeichnet, werden Instruktionen, deren Berechnung in jeder Schleifeniteration stets dasselbe Ergebnis liefert, aus der Schleife herausgezogen, da ihre wiederholte Ausführung keine neuen Informationen ergibt. Beim ***Superblock Loop Invariant Code Removal*** (SB-LICR) werden im Gegensatz dazu solche Anweisungen aus der Originalschleife entfernt, die nur auf dem Superblock selbst schleifeninvariant sind. Dazu wird vorausgesetzt, daß der Superblock die komplette Schleife vom Kopf bis zu einer Rücksprungkante durchquert. In diesem Fall sprechen wir auch von einer ***Superblock-Schleife***. Die entfernten Anweisungen werden dann in einen Pre-Header verschoben, der jedesmal ausgeführt wird, wenn die Ausführung den Superblock verlässt. Die simple Übertragung des SB-LICR von Chang [Chang et al., 1991] auf die High-Level-Ebene ist im Übergang von Abbildung 4.7(a) zu 4.7(b) dargestellt. Allerdings muß hierbei eine neue, äußere Schleife generiert werden, und die



(a) Ausgangscode für die Superblock LICR.



(b) Einfaches SB-LICR.



(c) Verbessertes SB-LICR

Abbildung 4.7.: Zwei Umsetzungen des High-Level SB-LICR .

Schleifengrenze (für die WCET-Analyse) für diese äußere Schleife muß sehr konservativ gewählt werden. Die neuen Schleifengrenzen allein ergeben daher eine erheblich *höhere* WCET. Dieses Verhalten lässt sich durch Flußbeschränkungen vermeiden, indem angegeben wird, daß das Verhältnis zwischen den WCECs des ersten Rumpfblocks der inneren Schleife und dem Block vor der äußeren Schleife durch die alte Schleifengrenze begrenzt wird. Dadurch wird die Anzahl der gemeinsamen Iterationen der Schleifen begrenzt. Eine einfachere Möglichkeit stellt jedoch das Verfahren dar, das in Abbildung 4.7(c) verwandt wurde. Hier wird der Pre-Header vor alle Nicht-Superblock-Rücksprungkanten kopiert und dadurch eine äußere Schleife vermieden. Falls die Schleifenbedingung komplexer ist oder Seiteneffekte aufweist, muß sie in beiden Verfahren gepuffert werden, um eine doppelte Auswertung zu vermeiden.

Die Anpassung an die WCET-Optimierung lässt sich also durch die beschriebenen Transformationen gewährleisten, allerdings wirft das SB-LICR einige weitere Probleme auf:

- Die reale Anwendbarkeit ist beschränkt: Beispiele wie das obere lassen sich natürlich kon-

struieren, kommen aber in der Realität kaum vor. Das klassische Low-Level-LICR verschiebt oftmals vom Codeselektor generierten Code, wie z.B. Arrayzugriffsberechnungen, aus der Schleife heraus. Diese Art von Code ist allerdings vollständig schleifeninvariant und nicht nur Superblock-schleifeninvariant und dazu teilweise erst auf der Low-Level-Ebene verfügbar.

- Da es bisher kein High-Level-LICR in der ICD-C IR gibt, fehlt ein Vergleichspunkt, um die Superblock-spezifischen Effekte zu bewerten.
- Für Schleifen, die nicht mindestens einmal ausgeführt werden, muß ein umschließendes `if` generiert werden. Dies führt zu Problemen, wenn die Schleife bereits von einem anderen Superblock umschlossen wird, da das neue `if` dann einen Einsprungpunkt in diesem Superblock generiert. Außerdem führt das neue `if` zu unproduktivem Codewachstum im umgebenden Code (siehe Konvertierung von `for`-Schleifen in Abschnitt 6.2).
- Falls der verschobene Code ursprünglich innerhalb einer Verzweigung stand, darf er keine Fehler/Traps auslösen können, da er nach der Verschiebung ***spekulativ*** ausgeführt würde. Da er in diesem Fall insgesamt potentiell öfter ausgeführt wird als vorher und dies mit evtl. neuen Eingabedaten, muß sichergestellt sein, daß er keine Programmabbrüche durch verursachte Fehler hervorruft. Damit sind bedingte Speicherzugriffe, Gleitkommaoperationen und Ganzzahldivisionen nicht verschiebbar.

Global Variable Migration

Die ***Superblock Global Variable Migration*** (SB-GVM) versucht analog zu den bisher vorgestellten Superblock-Optimierungen die klassische ***Redundant Load Elimination*** auf Superblöcke zu spezialisieren. Dazu werden Instruktionen gesucht, die einen Speicherzugriff durchführen, und es wird ermittelt, ob die Zieladresse des Zugriffs auf dem Superblock invariant ist (dafür muß sie nicht notwendigerweise global invariant sein). Falls ein oder mehrere solcher Zugriffe gefunden werden, werden diese durch Zugriffe auf eine neue, temporäre Variable ersetzt, die vor dem Superblock mit dem Wert der zugegriffenen Speicherstelle initialisiert wird und deren Inhalt an allen Superblock-Ausgängen wieder an die Speicherstelle zurückschreiben wird. Dadurch wird eine mehrfache, teure Adreßberechnung eliminiert. Besonders ergiebig ist diese Optimierung, analog zum SB-LICR, bei der Anwendung auf Superblock-Schleifen, allerdings ergeben sich hier dieselben Probleme wie beim SB-LICR. Dazu kommt, daß die ICD-C momentan keine High-Level Redundant Load Elimination anbietet, so daß auch hier wieder ein Vergleichspunkt fehlt.

Andere Optimierungen

Hwu [Hwu et al., 1993] führte weitere Superblock-spezifische Optimierungen ein, unter anderem das Verschmelzen zweier Superblöcke (***Branch Target Expansion***), um die Superblock-Länge und damit die Anzahl der für das Superblock-Scheduling verfügbaren Instruktionen zu erhöhen. Dies ist auf der High-Level-Ebene aber so nicht durchführbar, es entspricht dort dem Erzeugen eines Superblocks aus einem verlängerten Trace. Außerdem führt Hwu vor der Superblockbildung ebenfalls ein Loop Unrolling bzw. Loop Peeling, durch um die generierten Superblöcke zu verlängern. Diesen Ansatz werden wir ebenfalls verfolgen. Nach der Superblockbildung werden bei Hwu verschiedene Optimierungen durchgeführt (***Register Renaming***, ***Induction / Accumulator Variable Expansion***, ***Operation Migration / Combining***), um die Abhängigkeiten zwischen den Anweisungen des Superblocks zu verringern. Alle diese weiteren Optimierungen zielen jedoch darauf ab, eine Hilfe für das nachfolgende Scheduling zu sein, das in Rahmen dieser Arbeit nicht direkt mit der Superblockbildung verbunden ist.

5. WCEP-Neuberechnung

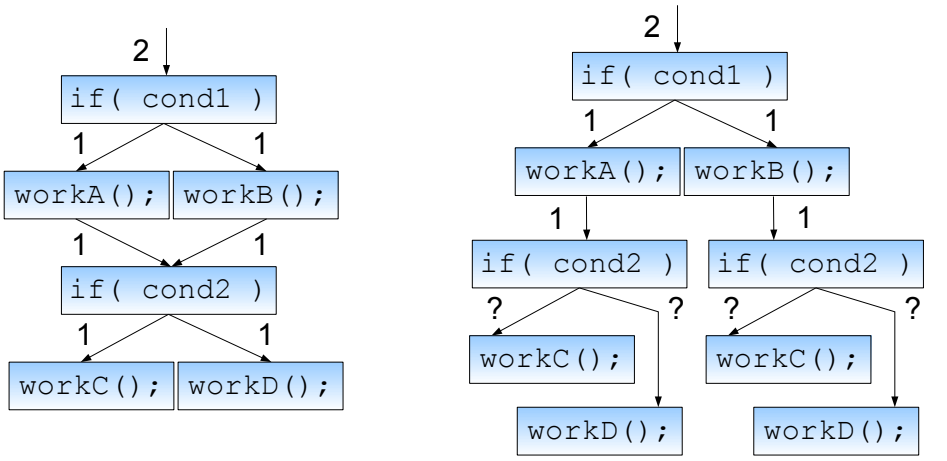
Im Zuge der WCET-Optimierungen taucht immer wieder das Problem auf, daß sichergestellt werden muß, daß die Optimierung noch immer auf dem WCEP operiert. Das zugrundeliegende Problem des Pfadwechsels wurde in Abschnitt 3.3 bereits vorgestellt. Die einfachste Lösung wäre, nach jedem Schritt mit einem sehr präzisen Werkzeug, wie z.B. aiT, den WCEP neu zu berechnen und dann zu überprüfen, ob man noch an der richtigen Stelle arbeitet. Dieses Vorgehen ist allerdings aus Effizienzgründen nicht realisierbar, weil bei größeren Benchmarks eine einzige, genaue Analyse mit aiT bereits mehrere Stunden dauern kann. Da im Verlauf einer Optimierungssequenz sehr viele Veränderungen am Code durchgeführt werden können, würde dieses Verfahren die Laufzeit des Kompilierprozesses in völlig inakzeptable Dimensionen treiben. Im WCC existierte deshalb bereits eine Möglichkeit, den WCEP auf LLIR-Ebene schnell neu zu berechnen. Dieses Neuberechnungsverfahren wurde im Rahmen der Diplomarbeit auf die ICD-C-Ebene übertragen und an die dortigen Gegebenheiten angepasst. Der Preis für die Beschleunigung der Analyse ist in beiden Fällen eine verringerte Genauigkeit. Das vorgestellte Verfahren ist daher nicht als Ersatz, sondern vielmehr als Ergänzung zu aiT zu verstehen.

5.1. Invariant Path

Der Invariant Path, der in 3.1.2 eingeführt wurde, könnte benutzt werden, um sicher zu gehen, daß *kein* Pfadwechsel vorliegt, denn solange die Superblöcke vollständig auf dem Invariant Path liegen, kann kein Pfadwechsel stattgefunden haben. Leider ist in der Praxis selbst hier eine Neuberechnung nötig, da durch die Superblockbildung die Kanten-WCECs ungültig werden, die aber für die nächste Traceselektion wieder gebraucht werden. Aus denselben Gründen, aus denen auch Pfadwechsel nicht lokal abschätzbar sind, ist auch die Änderung der Kanten-WCECs nicht lokal abschätzbar. Z.B. kann es sein, daß wir im Zuge der Superblockbildung ein komplettes `if` oder `while`-Statement kopieren mussten. Die Kanten-WCECs innerhalb dieses kopierten Konstrukts sind nicht mehr ohne eine globale WCET-Analyse aktualisierbar. Abbildung 5.1 stellt eine solche Situation, die eine globale Neuberechnung erfordert, dar. Die mit "?" gekennzeichneten Kanten-WCECs sind in der Situation nach der Superblockbildung, wie sie in Abbildung 5.1(b) dargestellt ist, nicht lokal aktualisierbar. Daher ist eine Neuberechnung innerhalb der Superblockbildung im Allgemeinen nicht möglich.

5.2. Aktualisierungen der Iteration-WCETs

Da wir in den Optimierungen teilweise Anweisungen aus einem Block löschen oder Ausdrücke in einer Anweisung ersetzen werden, müssen wir einen Weg finden, um die Iteration-WCET des einzelnen veränderten Blocks zu aktualisieren, da die Neuberechnung der globalen WCET die Iteration-WCETs der einzelnen Blöcke als gegeben voraussetzt. Wir verwenden hierfür eine einfache syntaxbaumbasierte Methode (s. Abschnitt 2.4.1). Den einzelnen Elementen des Syntaxbaums einer Anweisung oder eines Ausdrucks werden dabei Kosten zugewiesen und diese werden schließlich summiert. Dieses Verfahren berücksichtigt daher keine Abhängigkeiten der Laufzeit von Maschinenzuständen oder Programmkontexten. Dies ist auch auf der High-Level-Ebene so gut wie unmöglich, selbst auf der Low-Level-Ebene ist eine solche Worst-Case-Abschätzung des Maschinenzustands, wie sie auch in aiT integriert ist, nur mit hohem Aufwand möglich. In der High-Level-Ebene fehlen



(a) Codebeispiel mit Kanten-WCECs

(b) Selektierter Trace und die Kantengewichte des Longest-Path-Verfahrens

Abbildung 5.1.: Beispiel für lokal nicht aktualisierbare Kanten-WCECs

zusätzlich noch die Angaben über die ausgeführten Maschinenbefehle, was diese Aufgabe weiter erschwert. Daher bleiben dort eigentlich nur Heurstiken, um die Iteration-WCET abzuschätzen. Wir verwenden daher Kostentabellen zur Bestimmung der WCET und der Codegröße (CS) einer gegebenen Anweisung oder eines gegebenen Ausdrucks. Die WCET/CS von Basisblöcken wird dabei als Summe der WCET/CS der einzelnen Anweisungen berechnet und die WCET/CS einer Funktion ganz analog als Summe über die WCET/CS der Basisblöcke. Die dabei angenommene Kompositionalität der WCET ist in der Realität nicht direkt gegeben, die ermittelten Werte können aber immer noch als vernünftige Schätzung des WCET benutzt werden. Dabei sollte jedoch darauf geachtet werden, daß der Fehler mit steigender Anzahl der Iteration-WCET Aktualisierungen eher zunimmt. Deswegen werden wir in regelmäßigen Abständen eine erneute Analyse mit aiT durchführen, um diese Schätzfehler zu begrenzen.

Die verwendeten Kostentabellen sind in Tabelle 5.1, 5.2 und 5.3 dargestellt. Kursiv dargestellte Teile sind dabei Platzhalter für Unteranweisungen oder -ausdrücke. Alle dargestellten Konstanten wurden wie folgt ermittelt:

1. Ein Referenzprogramm ohne den gegebenen Ausdruck / die gegebene Anweisung wurde erstellt.
2. Die Iteration-WCET eines Basisblocks aus der Mitte des CFG wurde mithilfe von aiT bestimmt (kein Start- oder Endblock der Funktion).
3. Der untersuchte Ausdruck / die untersuchte Anweisung wurde in den gewählten Basisblock eingefügt.
4. Mittels aiT wurde erneut die Iteration-WCET des Basisblocks bestimmt.

Durch Vergleich der Ergebnisse der beiden WCET-Analysen wurde dann versucht, die Konstanten in den Gleichungen für die Iteration-WCET und die Codegröße festzulegen. Diese Prozedur wurde für mehrere Referenzprogramme durchgeführt, um den Meßfehler möglichst klein zu halten.

Die Experimente wurden mit dem TC1796 Prozessor als Zielarchitektur durchgeführt und sind daher auch nur dort nutzbar, auch wenn die Ergebnisse für den TC1797 wahrscheinlich vergleichbar sein werden. Da wir hier nur die *Iteration*-WCET abschätzen, wird bei Schleifen auch nur die

Anweisung	WCET (in Zyklen)	Codegröße (in Bytes)
`break;`	2	4
`continue;`	2	4
stmtA `;` *stmtB*	WCET(*stmtA*) + WCET(*stmtB*)	CS(*stmtA*) + CS(*stmtB*)
`do { } while (` *cond* `);`	2 + WCET(*cond*)	4 + CS(*cond*)
expression `;`	WCET(*expression*)	CS(*expression*)
`for (` *init* `;` *cond* `;` *inc* `; ) { }`	2 + WCET(*init*) + WCET(*cond*) + WCET(*inc*)	4 + CS(*init*) + CS(*cond*) + CS(*inc*)
`goto ...;`	1	4
`if (` *cond* `) { }`	3 + WCET(*cond*)	3 + CS(*cond*)
`return` *exp* `;`	2 + WCET(*exp*)	3 + CS(*exp*)
`switch (`*exp*`) { }`	$b <= 8$: $14 + b * 1$ + WCET(*exp*) $b > 8$: $22 + (b - 8) * 3$ + WCET(*exp*)	$b <= 8$: $2 + b * 4$ + WCET(*exp*) $b > 8$: $34 + (b - 8) * 8$ + WCET(*exp*)
`while (` *cond* `) { }`	2 + WCET(*cond*)	4 + WCET(*cond*)

Tabelle 5.1.: Kostentabelle für IR Statements

WCET der Schleifenbedingung und der Rücksprünge berücksichtigt, und nicht etwa die Laufzeit der gesamten Schleife. Im Falle der `switch`-Statements stellt die Konstante b die Anzahl der Case- und Default-Zweige dar. Die Laufzeit erhöht sich hier stark mit steigender Case-Zahl, da der WCC `switch`-Anweisungen bisher stets über Vergleichsabfolgen und nicht über eine Sprungtabelle umsetzt. Die Laufzeiten der Float-Anweisungen in Tabelle 5.3 wurden unter Verwendung der FPU des Tricore ermittelt.

5.3. High-Level IPET-Verfahren

Mit den gegebenen und durch die Optimierungen bei Bedarf aktualisierten Iteration-WCETs pro Basisblock werden wir den WCEP durch Aufstellung eines passenden ILP-Modells nach dem IPET-Verfahren neu berechnen. Dieses Verfahren wurde bereits in Abschnitt 2.4.3 kurz eingeführt. Wir folgen hier der Formulierung des ILP aus [Li & Malik, 1995], allerdings mit leichten Abweichungen. Im Gegensatz zu aiT verwenden wir *keine* Kontexte und verzichten auch auf die Erkennung unausführbarer Pfade, denn all dies würde eine intensive Vorarbeit in Form gründlicher Datenflußanalysen vorraussetzen, womit das Analysetool dann wieder eine ähnliche Komplexität (und damit auch Laufzeit) wie aiT aufweisen würde. Wir versuchen zum Ausgleich, soviel Informationen wie möglich aus den aiT-Analysen zu übernehmen, so z.B. die Iteration-WCETs und die Information, ob ein Block generell unausführbar ist (in diesem Fall können wir ihn von der WCEP-Suche ausschließen). Da die Schätzfehler bei der Aktualisierung der Iteration-WCETs mit steigender Zahl der Updates potentiell zunehmen, kann der Benutzer ein $k \in \mathbb{N}$ vorgeben, so daß bei jeder k-ten Neuberechnung eine komplette Analyse mit aiT vorgenommen wird, die die alten Iteration-WCETs überschreibt. Der Aufbau des Neuberechnungsmoduls ist in Abbildung 5.2 dargestellt, $i \in \mathbb{N}$ ist hierbei die Zahl der bisher durchgeführten WCEP-Neuberechnungen.

5.3.1. Aufbau des ILP

Wir werden im folgenden ein Integer Linear Program erstellen, das alle möglichen Kontrollflüsse im gegebenen Programm P modelliert und dessen Zielfunktionswert die Laufzeit des Programm ist. Diese Zielfunktion werden wir maximieren und somit die WCET des Programms erhalten. Zu

Ausdruck	WCET (in Zyklen)	Codegröße (in Bytes)

`a = b`
`a += b`
`a <<= b`
`a >>= b`
`a &= b`
`a |= b`
`a ^= b`
	$1 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$2 + \mathrm{CS}(a) + \mathrm{CS}(b)$

`a -= b`	$2 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$4 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a *= b`	$7 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$2 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a /= b` `a %= b`	$11 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$28 + \mathrm{CS}(a) + \mathrm{CS}(b)$

`a + b`
`a - b`
`a << b`
`a >> b`
`a & b`
`a | b`
`a ^ b`
	$1 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$2 + \mathrm{CS}(a) + \mathrm{CS}(b)$

`a < b`
`a > b`
`a <= b`
`a >= b`
`a == b`
`a != b`
	$1 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$4 + \mathrm{CS}(a) + \mathrm{CS}(b)$

`a * b`	$3 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$2 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a / b`	$9 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$22 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a % b`	$11 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$26 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a && b` `a || b`	$6 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$6 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a , b`	$\mathrm{WCET}(a) + \mathrm{WCET}(b)$	$\mathrm{CS}(a) + \mathrm{CS}(b)$
`f(` arg_1`, ...,` arg_n `)`	$2 + n + \sum_{i \in [1,n]} \mathrm{WCET}(arg_i)$	$4 + 4n + \sum_{i \in [1,n]} \mathrm{CS}(arg_i)$
comp`.`*field* *comp*`->`*field*	$1 + \mathrm{WCET}(comp) + \mathrm{WCET}(field)$	$2 + \mathrm{CS}(comp) + \mathrm{CS}(field)$
cond `?` *exp1* `:` *exp2*	$4 + \mathrm{WCET}(cond) +$ $\max\{\ \mathrm{WCET}(exp1), \mathrm{WCET}(exp2)\ \}$	$8 + \mathrm{CS}(cond) +$ $\mathrm{CS}(exp1) + \mathrm{CS}(exp2)$
base`[`*index*`]`	$12 + \mathrm{WCET}(base) + \mathrm{WCET}(index)$	$8 + \mathrm{CS}(base) + \mathrm{CS}(index)$
`+`*a* `-`*a* `*`*a*	$1 + \mathrm{WCET}(a)$	$2 + \mathrm{CS}(a)$
`!`*a* `~`*a*	$1 + \mathrm{WCET}(a)$	$4 + \mathrm{CS}(a)$
`++`*a* `--`*a*	$3 + \mathrm{WCET}(a)$	$4 + \mathrm{CS}(a)$
`&`*a*	$4 + \mathrm{WCET}(a)$	$4 + \mathrm{CS}(a)$
a`++` *a*`--`	$6 + \mathrm{WCET}(a)$	$10 + \mathrm{CS}(a)$
`(`*type*`)`*a*	$\mathrm{WCET}(a)$	$\mathrm{CS}(a)$
`sizeof(` *a* `)`	0	0
`(`*symbol*`)`	globle Symbole 2, andere 0	globle Symbole 6, andere 0

Tabelle 5.2.: Kostentabelle für IR Expressions (Integer Argumente)

Ausdruck	WCET (in Zyklen)	Codegröße (in Bytes)
`a = b`	$1 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$1 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a += b` `a -= b`	$3 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$6 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a *= b`	$4 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$6 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a /= b`	$16 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$4 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a + b` `a - b`	$3 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$4 + \mathrm{CS}(a) + \mathrm{CS}(b)$

`a < b`
`a > b`
`a <= b`
`a >= b`
`a == b`
`a != b`
	$2 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$8 + \mathrm{CS}(a) + \mathrm{CS}(b)$

`a * b`	$4 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$4 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a / b`	$16 + \mathrm{WCET}(a) + \mathrm{WCET}(b)$	$4 + \mathrm{CS}(a) + \mathrm{CS}(b)$
`a , b`	$\mathrm{WCET}(a) + \mathrm{WCET}(b)$	$\mathrm{CS}(a) + \mathrm{CS}(b)$
`+a`	$1 + \mathrm{WCET}(a)$	$1 + \mathrm{CS}(a)$
`-a`	$2 + \mathrm{WCET}(a)$	$4 + \mathrm{CS}(a)$
`++a` `--a`	$4 + \mathrm{WCET}(a)$	$8 + \mathrm{CS}(a)$
`&a`	$4 + \mathrm{WCET}(a)$	$4 + \mathrm{CS}(a)$
`a++` `a--`	$7 + \mathrm{WCET}(a)$	$10 + \mathrm{CS}(a)$
`(type)a`	$\mathrm{WCET}(a)$	$\mathrm{CS}(a)$
`sizeof( a )`	0	0

Tabelle 5.3.: Kostentabelle für IR Expressions (Float Argumente)

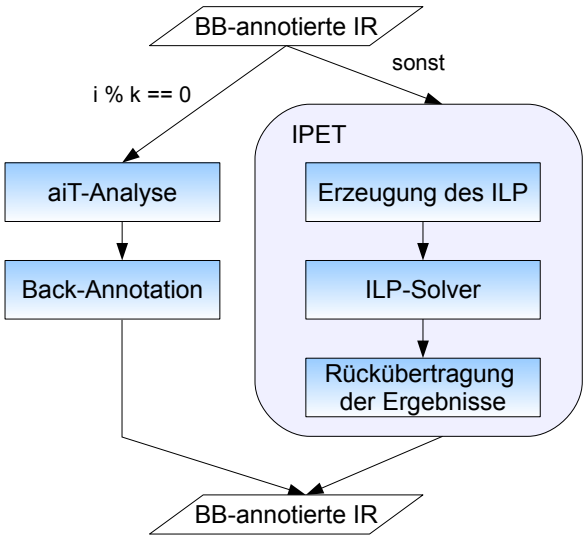


Abbildung 5.2.: Aufbau des Moduls zur Neuberechnung des WCEP.

einem gegebenen Kontrollflußgraphen $G = (V, E)$, wie er in 2.1 vorgestellt wurde, den ebenfalls gegebenen Iteration-WCET Werten c_b und den zu berechnenden Block-$WCEC_{sum}$ Werten x_b für $b \in V$ lautet die Zielfunktion:

$$\text{WCET}(P) = \max \sum_{b \in V \setminus V_{ref}} c_b \cdot x_b \tag{5.1}$$

V_{ref} ist hierbei die Menge der IR Basisblöcke, die Referenzblöcke nach Abschnitt 3.4.2 sind. Die Referenzblöcke tragen nicht zur WCET bei, da sie nicht einzeln, sondern immer nur im Verbund mit ihren referenzierten Blöcken ausgeführt werden können. Ohne weitere Nebenbedingungen ist der Lösungswert hierbei natürlich ∞. Daher werden wir im folgenden die einzelnen Nebenbedingungen vorstellen, die die x_b der einzelnen Basisblöcke so zueinander in Beziehung setzen, daß dadurch der mögliche Kontrollfluß widergespiegelt wird. Dabei werden wir noch einige weitere Variablen anlegen, um die Formulierung des ILP zu erleichtern, unter anderen auch Variablen, die die WCECs von Kanten darstellen. Diese tragen nicht zur Zielfunktion bei, sondern dienen nur dazu, das Verhältnis zwischen den $WCEC_{sum}$-Werten zweier Basisblöcke einfacher zu beschreiben. Die $WCEC_{sum}$-Werte von Knoten bzw. Blöcken i bezeichnen wir dabei durchgehend mit x_i und die WCEC-Werte von Kanten e bezeichnen wir mit d_e.

- Wir legen 2 neue Variablen $x_{cfg-start}$ und $x_{cfg-end}$ an, die einen virtuellen Eintritts- und Austrittspunkt des Kontrollflußgraphen darstellen. Dazu führen wir 2 Bedingungen ein, die festlegen, daß das Programm exakt einmal ausgeführt wird:

$$x_{cfg-start} = 1 \tag{5.2}$$
$$x_{cfg-end} = 1 \tag{5.3}$$

- Für jede Funktion f aus der Menge der Funktionen F legen wir eine Variable $x_{f-start}$ und eine Variable x_{f-end} als einen virtuellen Start- und Endknoten von f an. Wenn $b_{f-start}$ der Startblock der Funktion ist und $V_{f-return}$ die Menge der Blöcke mit `return` Statements, dann erzeugen wir die virtuellen Kanten $(x_{f-start}, b_{f-start})$ und $\forall v \in V_{f-return} : (v, x_{f-end})$ und die beiden Bedingungen:

$$x_{f-start} = d_{(x_{f-start}, b_{f-start})} \tag{5.4}$$
$$x_{f-end} = \sum_{v \in V_{f-return}} d_{(v, x_{f-end})} \tag{5.5}$$

  Falls f die `main`-Funktion ist, bei der die Programmausführung beginnt, so binden wir f über die virtuellen Kanten $(x_{cfg-start}, x_{f-start})$ und $(x_{f-end}, x_{cfg-end})$ und die folgenden Bedingungen an die CFG-Variablen an:

$$d_{(x_{cfg-start}, x_{f-start})} = x_{cfg-start} \tag{5.6}$$
$$d_{(x_{cfg-start}, x_{f-start})} = x_{cfg-end} \tag{5.7}$$

- Für jede Kante $e \in E$ im Kontrollflußgraphen legen wir eine neue Variable d_e an. Zu jedem Basisblock $b \in V$ legen wir eine Bedingung an, die die Flußerhaltung, auch ***Kirchhoff'sche Regel*** genannt, an diesem Knoten sicherstellt:

$$\sum_{v \in \delta^-(b)} d_{(v,b)} = \sum_{v \in \delta^+(b)} d_{(b,v)} \tag{5.8}$$

  Dabei sind δ^- und δ^+ auf Blöcke innerhalb derselben Funktion eingeschränkt. Falls b der Startblock einer Funktion f ist, addieren wir auf der linken Seite zusätzlich $d_{(x_{f-start}, b)}$, falls

$b \in V_{f-return}$ ist, addieren wir auf der rechten Seite $d_{(b,x_{f-end})}$. Wenn $b \in V_{ref}$ gilt und b_{ref} der referenzierte Block ist, so erzeugen wir zusätzlich noch die Bedingung

$$x_b = x_{b_{ref}} \tag{5.9}$$

da der Referenzblock und der referenzierte Block auf der LLIR-Ebene verschmolzen wurden und daher immer nur entweder zusammen oder gar nicht ausgeführt werden können. Falls die aiT-Analyse ergeben hat, daß der Block unausführbar ist, so berücksichtigen wir diese Information über die Bedingung

$$x_b = 0 \tag{5.10}$$

- Für jeden Basisblock, der eine Funktion g aufruft, erzeuge zwei virtuelle Kanten $(b, x_{g-start})$ und (x_{g-end}, b) für den Aufruf. Binde diese Kanten an den aufrufenden Block über die Bedingungen:

$$x_b = d_{(b,x_{g-start})} \tag{5.11}$$

$$d_{(b,x_{g-start})} = d_{(x_{g-end},b)} \tag{5.12}$$

  Außerdem müssen wir die Flußerhaltungsregeln für die virtuellen Knoten $x_{f-start}$ und x_{f-end} für alle Funktionen $f \in F$ ergänzen. Dazu betrachten wir alle bisher erzeugten virtuellen Kanten zu oder von diesen Knoten (Call/Return-Kanten sowie die Extra-Kanten für die `main`-Funktion) und addieren deren WCEC Werte analog zu Gleichung 5.8 auf.

- Für jede vorhandene Schleifengrenze (l,u), mit $l \in \mathbb{N}$ als unterer und $u \in \mathbb{N}$ als oberer Schranke für die Iterationszahl, bestimmen wir die Kantenmenge E_{leave}, die diejenigen Kontrollflußkanten umfasst, die die annotierte Schleife verlassen. Dies können `return`-Kanten oder Kanten zum Schleifennachfolgerblock sein. Danach spezifizieren wir das Verhältnis der WCECs von Schleifenkopfblock b_{head} und allen $e \in E_{leave}$. Für kopfgesteuerte Schleifen (`while` und `for`) generieren wir die Grenzbedingungen

$$x_{b_{head}} \geq l \cdot \sum_{e \in E_{leave}} d_e \tag{5.13}$$

$$x_{b_{head}} \leq (u+1) \cdot \sum_{e \in E_{leave}} d_e \tag{5.14}$$

  Da bei fußgesteuerten Schleifen (`do-while`) der Schleifenkopf nur so oft ausgeführt wird wie die Schleife selbst, erzeugen wir dort eine leicht modifizierte Bedingungskombination

$$x_{b_{head}} \geq l \cdot \sum_{e \in E_{leave}} d_e \tag{5.15}$$

$$x_{b_{head}} \leq u \cdot \sum_{e \in E_{leave}} d_e \tag{5.16}$$

  Li & Malik [Li & Malik, 1995] haben hier im Unterschied zu uns die WCEC des Blocks vor der Schleife und des ersten Blocks des Schleifenkörpers ins passende Verhältnis gesetzt. Dies wirft allerdings Probleme auf, wenn zwei Schleifen direkt aufeinander folgen.

- Abschließend generieren wir für jede Flußbeschränkung (s. Abschnitt 3.1.4) der Form $k_{l1}x_{l1} + k_{l2}x_{l2} + ... + k_{ln}x_{ln} \leq k_{r1}x_{r1} + k_{r2}x_{r2} + ... + k_{rm}x_{rm}$ $(\forall k_i : k_i \in \mathbb{N})$ ihr direktes Äquivalent als ILP-Bedingung. Da die Flußbeschränkungen explizit die WCECs von Basisblöcken über lineare Ungleichungen zueinander in Bezug setzen, ist die Übertragung hier 1:1 möglich.

5.3.2. Lösung des ILP-Modells

Das generierte ILP-Modell wird danach von der im WCC-Framework bereits bestehenden LIBILP Biliothek gelöst. Hierzu stehen intern verschiedene ILP-Solver zur Verfügung, wie z.B. der kostenlose Solver `lp_solve` [lp_solve, 2009], der auf der bekannten Simplex-Methode aufbaut und für die Lösung von ILPs Branch-and-Bound-Techniken anwendet, sowie der proprietäre Solver CPLEX [ILOG, 2009] der sowohl die Simplex-Methode als auch neuere Verfahren wie z.B. das Interior-Point-Verfahren beherrscht und lineare, ganzzahlig lineare und quadratische Optimierungsprobleme lösen kann.

5.3.3. Rückübertragung der Ergebnisse

Die Rückübertragung der Ergebnisse ist aufgrund der gewählten Formulierung relativ einfach. Pro Basisblock b übernehmen wir die folgenden Ergebnisse in das `IR_WCETObject` des Basisblock:

$$\mathrm{WCEC}_{sum}(b) = x_b \tag{5.17}$$

$$\mathrm{WCET}_{sum}(b) = x_b * c_b \tag{5.18}$$

$$\forall v \in \delta^+(b) : \mathrm{WCEC}_{sum}((b,v)) = d_{(b,v)} \tag{5.19}$$

Die WCET_{sum} Werte von Funktionen, Übersetzungseinheiten und der gesamten IR lassen sich danach aus den Summen der WCETs der in ihnen enthaltenen Basisblöcke zusammensetzen. Mit dem gegebenen ILP ist es nicht möglich, WCECs von Funktionsaufrufkanten zu rekonstruieren, da wir aus den Ergebnissen nicht extrahieren können, von welchem `return`-Statement der aufgerufenen Funktion wir wieder zurückspringen. Da wir diese Information aber an keiner Stelle benötigen, spielt dies keine Rolle.

6. WCET-gesteuerte High-Level Superblockbildung

Anders als die bekannten Optimierungen wollen wir die Superblockbildung durch WCET-Daten steuern und auf der High-Level Ebene anwenden, um frühestmöglich weiteres Optimierungspotential freizusetzen. Der dazu verwendete Algorithmus wird im folgenden vorgestellt. Er weicht an vielen Stellen von den bisher verwendeten, relativ einfachen Low-Level Algorithmen ab. In Abschnitt 6.1 werden wir das Rahmenwerk kennenlernen, das die einelnen Module der Superblock-Bildung und -Optimierung miteinander verbindet, in Abschnitt 6.2 wird ein Modul vorgestellt, das einige vorbereitende Transformationen vornimmt, in Abschnitt 6.3 werden die verwendeten Traceselektionsalgorithmen präsentiert und in Abschnitt 6.4 wird der Algorithmus für die High-Level-Superblockbildung dargestellt.

6.1. Rahmenwerk

Der Ablauf der Superblockoptimierung wird in Abbildung 6.1 dargestellt. Der erste Schritt besteht aus vorbereitenden Transformationen, die wir den ***Prepass*** nennen (s. Abschnitt 6.2). Danach werden in dieser Reihenfolge die Alias-Analyse (s. Abschnitt 7.1.2) und die initiale WCET-Analyse durchgeführt. Daraufhin wird die Hauptschleife betreten, die für alle Funktionen, absteigend nach ihrer $WCET_{sum}$ sortiert, die Superblockbildung und -optimierungen aufruft, bis alle Basisblöcke der Funktion von einem Superblock überdeckt werden. Wir sortieren die Funktionen hierbei nach ihrer WCET, damit die Funktionen, die am meisten zur WCET des Programms beitragen, zuerst abgearbeitet werden. Da wir in Abschnitt 6.3 unter anderen den Codegrößenzuwachs des gesamten Programms beschränken werden, ist es wichtig, daß die Funktionen in dieser Reihenfolge abgearbeitet werden, da sonst das zulässige Codewachstum bereits durch Funktionen erschöpft sein kann, die nur wenig zur WCET beitragen. Pro Funktion wird dabei iterativ jeweils ein Trace selektiert (s. Abschnitt 6.3), aus dem Trace wird ein Superblock gebildet (s. Abschnitt 6.4), dieser Superblock wird unter Anwendung der implementierten Superblock-Optimierungen optimiert (s. Kapitel 7) und abschließend wird der WCEP neu berechnet (s. Kapitel 5). Die Neuberechnung wird ausgelassen falls die Superblock-Bildung und -Optimierungen den Code nicht verändert haben. Dies kommt insbesondere in den letzten Iterationen der Hauptschleife häufiger vor, weil die noch verfügbaren Traces immer kürzer werden. SB-CSE und SB-DCE sind einzeln an- und abschaltbar und benutzen die in Kapitel 7 ebenfalls vorgestellten neu eingeführten Def/Use-Sets und Liveness-Informationen. Damit die einzelnen Schritte durchführbar sind und die WCET-Annotationen dabei nicht verloren gehen, müssen diese, wie in Abschnitt 3.4.2 erläutert, an den passenden Stellen von den `IR_BasicBlock`-Objekten an die `IR_Stmt`-Objekte umgehängt werden. Die Traceselektion und die Neuberechnung des WCEP arbeiten dabei auf den Basisblock-basierten WCET-Annotationen, der Rest der Module (diejenigen, die den Code verändern) arbeitet auf den Statement-basierten Annotationen. Abschließend wird die "Polyhedral If-Statement Optimization" aufgerufen (in Abbildung 6.1 als "Post SB-Optimierungen" gekennzeichnet) falls diese in der aktuellen Konfiguration aktiviert ist (standardmäßig ab Optimierungslevel O3 der Fall). In manchen Fällen können hierdurch die von der Superblockbildung erzeugten Schachtelungen von `if`-Anweisungen komprimiert werden, indem Schachtelungen mit sich widersprechenden `if`-Bedingungen erkannt und aufgelöst werden.

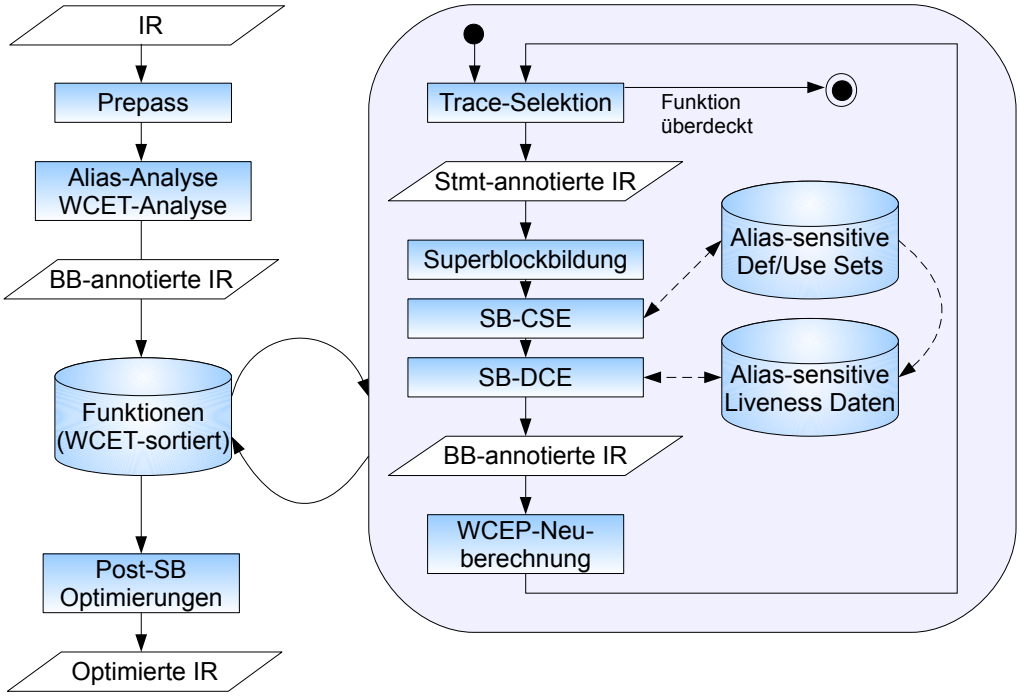


Abbildung 6.1.: Aufbau der High-Level Superblock-Optimierungen.

6.2. Prepass

Der Prepass formt das Eingabeprogramm so um, daß einige Codekonstrukte, die die Formulierung der nachfolgenden Superblockalgorithmen erschweren würden, eliminiert werden. Außerdem werden einige der Standard-Optimierungen, die solche Codekonstrukte generieren, in kontrollierter Form gestartet, falls sie beim Aufruf des WCC aktiviert wurden. Der genaue Ablauf des Prepass ist:

1. ICD-C Function Inlining (eingeschränkt)
2. ICD-C Function Unrolling (eingeschränkt)
3. Für jede Funktion:
   a) Versuche, vorhandene `gotos` zu eliminieren
   b) Test auf `gotos` und unstrukturierte `switch`-Anweisungen
      i. Treffer:
      Superblockoptimierungen abbrechen
      ii. Sonst:
      Einebnung verschachtelter `IR_CompoundStmts`
      Ersetzung von `for`-Schleifen
      Auflösung von bedingten Ausdrücken (falls aktiviert)
      `switch`-Statement Anpassungen

ICD-C Loop Unrolling und Function Inlining

Um der Superblockbildung größere, zusammenhängende Codeabschnitte zu liefern, auf denen sie arbeiten kann, ruft der Prepass das Loop Unrolling und Function Inlining auf. Hierbei ist jedoch zu beachten, daß beide Optimierungen `gotos` erzeugen können. Beim Unrolling ist dies der Fall,

```
send( short *to, short *from, int count)
{
  int n = ( count + 3 ) / 4;
  switch( count % 4 ){
    case 0: do { *to++ = *from++;
    case 3:      *to++ = *from++;
    case 2:      *to++ = *from++;
    case 1:      *to++ = *from++;
            } while ( --n > 0 );
  }
}
```

Abbildung 6.2.: Duff's Device: Ein Beispiel für unstrukturierten Code.

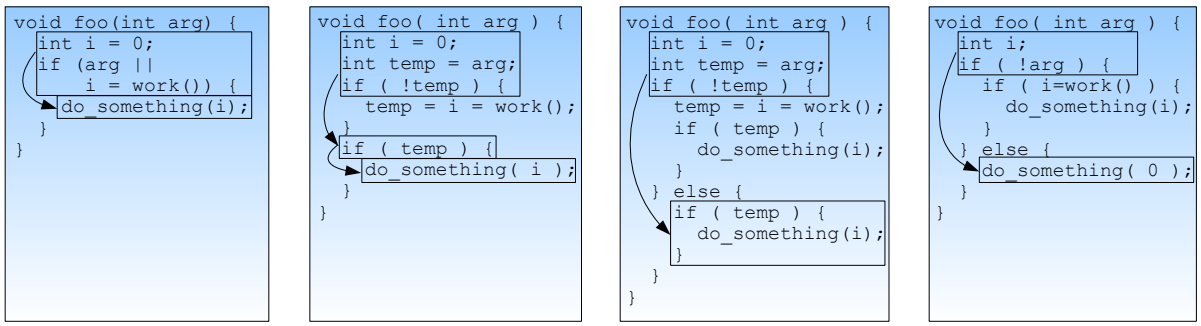
falls die abgerollte Schleife `continue`s beinhaltet (für den Sprung zum nächsten Schleifenkopf), und beim Inlining ist es immer der Fall (für den Sprung von der `return`-Anweisung zum Ende der eingesetzten Funktion), aber nur wenn die eingesetzte Funktion mehr als ein `return`-Statement hat, führt dies zu Problemen. Falls nur ein `return` vorhanden ist, kann der Algorithmus aus dem nächsten Abschnitt das erzeugte `goto` eliminieren. Daher steuert der Prepass die beiden Optimierungen so, daß keine nicht eliminierbaren `goto`s erzeugt werden. Aus diesem Grund wird, falls die Superblockbildung aktiviert ist, die ICD-C Tail Recursion Elimination grundsätzlich ausgeschaltet, da diese eben solche `goto`s generiert.

Test auf `goto`s und unstrukturierte `switch`-Anweisungen

Wir betrachten eine `switch`-Anweisung als ***unstrukturiert***, wenn nicht alle ihre `case` und `default`-Markierungen im selben `IR_CompoundStmt` liegen. Ein Beispiel hierfür ist Duff's Device (siehe [Duff, 1983] bzw. Abbildung 6.2). Da die Algorithmen zur Superblockbildung strukturierten Code vorraussetzen, muß der Prepass die Existenz von `goto`s und unstrukturierten `switch`s überprüfen und die betroffenen Funktionen von der Superblockoptimierung ausschließen. Im aktuellen Prepass wird davor versucht, `goto`s zu entfernen, die zu ihren direkten Nachfolger-Statement springen und die daher keinen Effekt haben. Solche `goto`s werden oft von der ebenfalls im Prepass aufgerufenen Optimierung Function-Inlining erzeugt. Erosa beschreibt in [Erosa & Hendren, 1994] einen Ansatz zur vollständigen Entfernung von unstrukturiertem Code aus Programmen, allerdings ist die erzielte Performance fragwürdig, daher verwenden wir nur das oben erwähnte simple Verfahren. Falls dieses alle `goto`s eliminieren kann und keine unstrukturierten `switch`s vorhanden waren, so werden auch alle Sprungmarken gelöscht. Falls nicht, wird die Superblockoptimierung für diese Funktion nicht fortgesetzt. Funktionen mit unstrukturiertem Code sind allerdings selbst in Code für eingebettete Systeme sehr selten: In der Testbench des WCC mit mittlerweile über 100 Benchmarks aus den verschiedensten Embedded-Testsuiten (siehe Anhang A) sind lediglich 3 Benchmarks mit unstrukturiertem Code enthalten.

Einebnung verschachtelter `IR_CompoundStmt`s

Anweisungen der Form `while (c) { i++; { foo(i); } }` werden vereinfacht. Im Beipiel wäre das Ergebnis: `while (c) { i++; foo(i); }`. In den inneren Compounds deklarierte Variablen werden dabei in die äußeren verschoben und bei Bedarf umbenannt. Dies hat den Zweck, daß die Anweisungen eines Basisblocks danach innerhalb desselben Compounds liegen, was die Bildung der High-Level-Superblöcke vereinfacht.



(a) Original-Code (b) Bedingter Ausdruck ersetzt (c) Superblock gebildet (d) Optimierter Code

Abbildung 6.3.: Transformation von bedingten Ausdrücken im Prepass.

Ersetzung von `for`-Schleifen

`for`-Schleifen stellen in den Optimierungen an vielen Stellen einen Speziallfall dar, weil sie als einzige High-Level-Anweisung nicht eindeutig einem Basisblock zugeordnet sind. Daher werden sie im Prepass transformiert, nach Möglichkeit in `do-while`- und sonst in `while`-Schleifen. `do-while`-Schleifen werden dabei bevorzugt erzeugt, weil bei diesem Schleifentyp der Abbruch der Schleife nicht mit einem Sprung verbunden ist. Die ICD-C Optimierung "Transform Head Controlled Loops" wandelt daher explizit alle `for`- und `while`-Schleifen in `do-while`-Schleifen um, sofern das bei der jeweiligen Schleife möglich ist. Der Prepass geht bei der Umwandlung einer `for`-Schleife `for ( <init>; <cond>; <inc> ) { <body> }` folgendermaßen vor:

1. Der `<init>`-Teil wird inklusive aller enthaltenen Variablendeklarationen vor die Schleife verschoben.
2. Durch den in der ICD-C vorhandenen Loop-Analyzer wird festgestellt, ob `<body>` immer mindestens einmal ausgeführt wird. Falls ja, lege eine neue `do-while`-Schleife mit gleichem Rumpf und gleicher Abbruchbedingung an, sonst eine äquivalente `while`-Schleife.
3. Kopiere die mit der Schleife verknüpften Flowfacts an die neue Schleife
4. Kopiere den `<inc>`-Teil vor jede Rücksprungkante zum Schleifenkopf (bei `while`) oder integriere ihn per Kommaoperator in die Schleifenbedingung, die dann `<inc>, <cond>` lautet (bei `do-while`).
5. Ersetze die `for`-Schleife durch die neu generierte Schleife.

Auflösung von bedingten Ausdrücken

Bedingte Ausdrücke, die mit Hilfe der `&&`, `||` oder `:?` Operatoren ([ISO/IEC, 1999]) gebildet wurden, werden auf IR-Ebene durch einen einzelnen Basisblock überdeckt, auf der LLIR Ebene jedoch durch mehr als einen Block (1:n-Beziehung, s. Abschnitt 3.4.2). Da die Traceselektion und die Superblockoptimierungen nur auf Basisblockebene arbeiten, müssen diese stets die Abhängigkeiten und die WCET-Anteile der bedingt ausgeführten Ausdrücke mit berücksichtigen. Dies kann zu ungünstigen Annahmen führen, insbesondere dann, wenn die bedingten Ausdrücke Schreibzugriffe enthalten, wie z.B. in Abbildung 6.3(a) dargestellt. Die Ersetzung des bedingten Ausdrucks ist in 6.3(b) dargestellt, in 6.3(c) wurde ein Superblock gebildet und in 6.3(d) wurden die Standard-Optimierungen Value Propagation und Dead Code Elimination bzw. Unused Symbol Elimination

benutzt, um den Code weiter zu vereinfachen. Im Unterschied zur Situation in 6.3(a) ist die Zuweisung `i = 0` nun tot und der Aufruf `do_something(0)` ist für die ICD-C Function Specialization zugänglich geworden.

Das in Abbildung 6.3 bereits illustrierte Verfahren zur Ersetzung bedingter Ausdrücke wird im folgenden kurz skizziert. Für weiteren Erläuterungen benötigen wir noch einige Definitionen: Die Menge der im Programm vorhandenen `IR_Exp` Objekte nennen wir X. Die Hierarchie-Relation $<_{edirect} \subset X \times X$ ist induktiv definierbar: $(e_i, e_j) \in <_{edirect}$ gilt dabei genau dann, wenn e_i ein direkter Teilausdruck von e_j ist, wie bei den Ausdrücken `a` und `a == b`. Die reflexiv-transitive Hülle von $<_{edirect}$ nennen wir $\leq_e$. Wir nennen einen Ausdruck $e \in X$ genau dann ***bedingt***, wenn e eine `IR_BinaryExp` mit Operator `&&` oder `||` ist, oder wenn e eine `IR_CondExp` ist (also dem `:?` Operator entspricht). Die Menge der bedingten Ausdrücke wird als $X_b \subset X$ bezeichnet.

1. Der Benutzer kann zwischen 3 Ersetzungsmodi wählen: Keine Ersetzungen, Ersetzungen bei bedingten Schreibzugriffen und Ersetzung aller bedingten Ausdrücke. Je nachdem welcher dieser Modi gewählt wurde, werden zuerst die Ausdrücke gesucht, auf die das gewählte Kriterium zutrifft. Für die Erkennung eines Schreibzugriffs benutzen wir dabei die in Abschnitt 7.1.3 noch vorzustellenden Def/Use-Sets, mit denen wir überprüfen können, ob ein Ausdruck $e \in X_b$ überhaupt einen Schreibzugriff enthält, nämlich genau dann, wenn $\text{DEF}_{\text{may}}(e) \neq \emptyset$ ist. $\text{DEF}_{\text{may}}(e)$ ist, wie wir in Abschnitt 7.1.3 näher erläutern werden, hierbei die Menge der Symbole, die durch den Ausdruck e beschrieben werden können.

2. Für jeden gefundenen, zu ersetzenden Ausdruck $e \in X_b$ finde den Ausdruck $e_{top} \in X_b$, so daß gilt $e \leq_e e_{top} \wedge \nexists e_{higher} \in X_b : e \leq_e e_{higher} \wedge e_{top} \leq_e e_{higher}$. Bei diesem höchstgelegenen bedingten Ausdruck starten wir die Konvertierung durch Aufruf von `expressionBFS(`e_{top}`)`, damit die Auswertungsreihenfolge erhalten bleibt.

3. `expressionBFS(`$e \in X$`)`: Falls e bedingt ist, rufe `expandCondExp(`e`)` auf, ansonsten rufe `expressionBFS(`e_{child}`)` für alle $e_{child} <_{edirect} e$ auf.

4. `expandCondExp(`$e \in X_b$`)`: Falls e keine Schleifenbedingung ist, rufe `transform(`e`)` auf, sonst:
   a) Erzeuge eine neue Variable `t`.
   b) Erzeuge neue Anweisungen der Form `t =` $copy(e)$`;` die vor jedem `continue`, ans Ende des Schleifenrumpfes und vor die Schleife (letzteres nur bei `while`-Schleifen) platziert werden. Rufe für jede der Kopien `transform(`$copy(e)$`)` auf.
   c) Ersetze e durch eine Auswertung von `t`.

5. `transform(`$e \in X_b$`)`: Erzeuge eine neue Variable `t` sowie neuen Code und füge diesen vor der Anweisung ein, zu der e gehört. Der Code unterscheidet sich dabei je nach dem verwendeten e:
   a) $e =$ `a && b` $\longrightarrow$ `t = a; if(t) { t = b; }`
   b) $e =$ `a || b` $\longrightarrow$ `t = a; if(!t) { t = b; }`
   c) $e =$ `a ? b : c` $\longrightarrow$ `if(a) { t = b; } else { t = c; }`

   Führe die Ersetzungen im neu generierten Code fort, bis keine zu ersetzenden bedingten Ausdrücke mehr gefunden werden.

`switch`-Statement Anpassungen

Um den Kontrollfluß innerhalb von `switch`-Statements explizit sichtbar zu machen, wird jedem `switch`, das noch kein `default`-Statement hat, ein solches ans Ende des `switch`-Compounds angehängt. Außerdem wird das `switch`-Compound durch ein neu erzeugtes `break` beendet, falls es

```
void foo(int arg) {
  switch(arg) {
    case 0:
       ...; return;
    case 1:
       ...; return;
    case 2:
       arg++;
  }
  return;
}
```

(a) Original-Code

```
void foo(int arg) {
  switch(arg) {
    case 0:
       ...; return;
    case 1:
       ...; return;
    case 2:
       arg++; break;
    default:
       break;
  }
  return;
}
```

(b) Angepasstes `switch`-Statement

Abbildung 6.4.: Anpassung von `switch`es im Prepass.

sonst einen Fall-Through-Pfad zu dem Statement, das dem `switch` folgt, gäbe. Ansonsten kann es vorkommen, daß, wie in Abbildung 6.4(a) skizziert, sich die Anweisungen aus einer Verzweigung des `switch` (hier das `arg++`) im selben Basisblock befinden wie die dem `switch` nachfolgenden Anweisungen (hier das `return`). Um diesen Spezialfall auszuschließen und um sicherzugehen, daß jeder `switch`-Zweig durch ein `break;` beendet wird, nehmen wir die beschriebene Transformation vor.

6.3. Traceselektion

Die bekannten Verfahren zur Traceselektion wurden in Abschnitt 4.1 bereits vorgestellt. Wir werden jedoch auch ein neues Verfahren testen, das im folgenden vorgestellt werden soll. Der Ausgangspunkt für die Traceselektion ist, daß wir bereits bestimmt haben, wo der WCEP liegt. Die Traces werden dann nur noch aus Blöcken gebildet, die auf dem WCEP liegen. Auf den ersten Blick scheint dies die Traceselektion trivial werden zu lassen, allerdings ist dem nicht so, da fortgeschrittene Analysewerkzeuge, wie z.B. das bei uns verwendete aiT, das in Abschnitt 2.6 vorgestellte Konzept der Kontexte unterstützen. Der WCEP ist bei solchen, genaueren Analysen kein einfacher Pfad durch die betrachtete Funktion, sondern es kann vorkommen, daß *mehrere* Ziele einer Verzweigung auf dem WCEP liegen, und zwar in jeweils unterschiedlichen Schleifendurchläufen oder Funktionsaufrufen. Daher müssen wir in solchen Fällen entscheiden können, welche Zweige wir für die Traces wählen. Im nächsten Abschnitt werden wir zeigen, daß die bestehenden Heuristiken dafür in einigen Fällen schlecht geeignet sind. Zuvor wollen wir kurz darauf eingehen, wie die Traces in der IR dargestellt werden.

Wir nutzen zur Speicherung der bestehenden Traces / Superblöcke die neue Annotationsklasse `IR_SuperblockInfo`, die in Abbildung 6.5 dargestellt ist. Diese ist ebenso wie die bestehenden Annotationsklassen (z.B. `IR_WCETObject`) als Statement- oder Basisblockannotation handhabbar und zwischen diesen beiden Darstellungen konvertierbar. Die in ihr gespeicherten Informationen sind zum einen eine eindeutige Trace-ID, der Trace-Index, und die Nachfolge- und Vorgänger-Statements oder Basisblöcke, je nachdem in welchem Modus die Annotation gerade benutzt wird. Die Traces werden also nicht zentral verwaltet, sondern für jeden Basisblock wird seine Zugehörigkeit zu einem Trace nur lokal in seinem `IR_SuperblockInfo`-Objekt hinterlegt. Der Trace-Index gibt dabei auch an, ob der Block überhaupt auf einem Trace liegt, alle Werte über 0 werden an erzeugte Traces vergeben, ein Wert von 0 ist daher nur bei Blöcken vorhanden, die noch auf keinem Trace liegen. Außerdem enthält die Klasse einige Hilfsmethoden, z.B. um festzustellen ob zwei Blöcke auf demselben Trace liegen, und Methoden zum Setzen der Nachfolger- und Vorgängerbe-

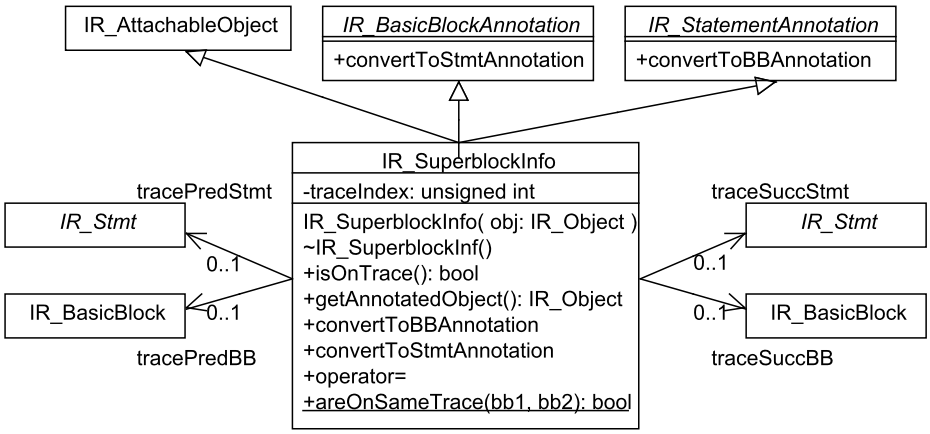


Abbildung 6.5.: Aufbau der `IR_SuperblockInfo`-Klasse.

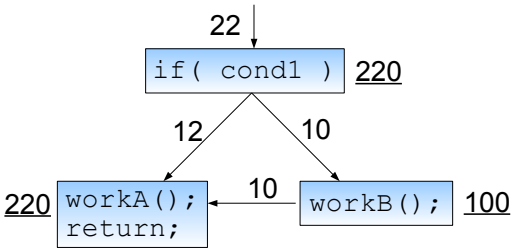


Abbildung 6.6.: Beispiel: Fehler der Kantengewichts-basierten Selektion.

ziehungen (im Diagramm nicht aufgeführt). Die Erzeugung der `IR_SuperblockInfo`-Objekte findet erst statt, wenn die Traceselektion abgeschlossen ist und der gewählte Trace in der IR gespeichert wird.

6.3.1. Mängel der bekannten Verfahren

Prinzipiell sind verschiedene Ziele bei der Selektion denkbar. Da wir nur WCEP-Blöcke selektieren, erhalten wir in jedem Fall einen Trace, dessen Optimierung zur WCET-Minimierung beiträgt. Darüber hinaus sind folgende Selektionsziele möglich:

- **Selektion des am häufigsten ausgeführten Pfades**: Dies ist ein typisches ACET-Kriterium, daher ist die Kantengewichts-basierte Selektion hier auch eine gute Heurisik, deren Überlegenheit gegenüber der Knotengewichts-basierten Selektion in Abschnitt 4.1 bereits präsentiert wurde. Sie garantiert uns aber nicht, daß der gewählte Pfad wirklich in dieser Form ausführbar ist, denn die Edge-WCECs, auf deren Basis wir diese Heuristik ausführen würden, können über mehrere Schleifeniterationen oder mehrere Funktionsaufrufe kumuliert worden sein. Da wir die Selektion nur auf Basisblock-WCETs und Edge-WCECs stützen können, tritt dieses Problem bei allen nachfolgend genannten Algorithmen auf, ebenso wie bei allen Algorithmen, die auf Profiling-Daten aufbauen, da auch diese über viele verschiedene Profilingläufe gemittelt oder summiert werden müssen.
- **Traceselektion nach Optimierungspotential**: Eine weitere Alternative, die z.B. von Hank in [Hank et al., 1993] teilweise auch umgesetzt wird, ist die Selektion nach (vermu-

tetem) Optimierungspotential. Dabei werden Blöcke, deren Effekte besser abschätzbar sind (z.B. keine Speicherzugriffe, keine Funktionsaufrufe, keine Synchronisierungsinstruktionen), bei der Selektion bevorzugt. Das Verfahren in [Hank et al., 1993] ist allerdings großteils Low-Level-spezifisch und dient auch in Teilen dazu, dort fehlende Compilerelemente wie eine Alias-Analyse oder interprozedurale Analyse auszugleichen. Daher ist es nicht direkt auf die High-Level-Anwendung übertragbar. Das Optimierungspotential ist im Allgemeinen noch schwieriger einzuschätzen als der Effekt einer Optimierung, der selbst bereits nicht mehr berechenbar ist.

- **Selektion des Pfades mit der größten Gesamt-WCET**: Als WCET eines Pfades betrachten wir hierbei die summierten $WCET_{sum}$-Werte seiner Basisblöcke. In Anlehnung an den Begriff der Ausführungs*dauer* werden wir den Pfad mit der größten Gesamt-WCET auch als ***längsten Pfad*** bezeichnen. Von der Selektion dieses Pfades erhoffen wir uns, daß wir auf ihm die WCET am stärksten verbessern könne, da er auch am stärksten zu ihr beiträgt. In Abbildung 6.6 ist ein Beispiel abgebildet, das zeigt, daß die Kantengewichts-basierte Selektion in diesem Fall ungünstige Ergebnisse liefern kann. Die Knoten-$WCET_{sum}$ ist dort zur Unterscheidung von der Kanten-$WCEC$ unterstrichen dargestellt. Die Kantengewichts-basierte Selektion würde in diesem Beispiel ebenso wie die Knotengewichts-basierte den Pfad zum linken Knoten wählen, mit einer WCET von $220+220=440$ Zyklen, während der Pfad über den rechten Knoten mit $220+100+220=540$ Zyklen der längste Pfad im Graphen ist. Da dies ein realitätsnahes Beispiel war, werden wir in 6.3.2 eine fortgeschrittene Selektion entwickeln, die immer den längsten Pfad findet.

6.3.2. Longest-Path Verfahren

Im folgenden wird das für die High-Level-Traceselektion vorrangig verwendete Longest-Path Verfahren vorgestellt. Dieses Verfahren trifft seine Entscheidungen ebenfalls auf Grundlage der vorhandenen Kanten-$WCECs$ und Knoten-$WCET_{sum}s$, wählt dabei aber keinen einfachen greedy-Ansatz wie die bestehenden Heuristiken, sondern bestimmt den längsten Pfad über einen Graphalgorithmus. Dazu wird im Einzelnen wie folgt vorgegangen:

1. Wie in den bekannten Heuristiken wird ein Startknoten b_{seed} ausgewählt. Wir wählen hier stets den Knoten mit maximaler $WCET_{sum}$ unter allen Knoten mit Trace-Index 0.

2. Wir selektieren den Trace inherhalb der Schleife L, die b_{seed} umgibt oder im `IR_CompoundStmt` des Funktionsrumpfes, auch Top-Level-Compound genannt, falls eine solche Schleife nicht existiert. Wir beschreiben das Vorgehen im folgenden für den Fall einer umgebenden Schleife, der Fall der Selektion im Top-Level-Compound funktioniert analog. Innerhalb dieser Schleife L bauen wir einen gerichteten, azyklischen Graphen $G_L = (V_L, E_L)$ auf, konkret nutzen wir dafür die `boost`-Graphbibliothek [Boost, 2009].

   a) Analog zu den `IR_Exp`-Objekten (s. Abschnitt 6.2) lässt sich auch auf der Menge S der `IR_Stmt`-Objekte eine Hierarchie-Relation $\leq_s$ als reflexiv-transitive Hülle einer direkten Enthaltenseinsrelation $<_{sdirect}$ definieren. Wir definieren die ***Schleifentiefe*** einer Anweisung $s_i \in S$ als Kardinalität der Menge $\{s_j | s_i \leq_s s_j \wedge s_j \text{ ist Schleife}\}$. Es lässt sich leicht zeigen, daß mit dieser Definition alle Anweisungen eines Basisblocks dieselbe Schleifentiefe haben müssen, daher definieren wir die Schleifentiefe eines Blocks als die seiner Anweisungen.
   Zur Knotenmenge V_L addieren wir nun die Basisblöcke b des WCEPs, in denen für alle Statements $s \in b$ die Eigenschaft $s \leq_s L$ gilt (auch diese Eigenschaft gilt entweder für alle oder für kein Statement eines Blocks) und die dieselbe Schleifentiefe wie L haben. Für jede innere Schleife I mit $I \leq_s L$ und Schleifentiefe(I) = Schleifentiefe$(L)+1$, die auf dem WCEP liegt, legen wir einen Knoten b_{iloop_I} an, der den Kopf der inneren

Schleife repräsentiert. Die Menge dieser Knoten, die innere Schleifen darstellen, nennen wir V_{iloop}. Wir bezeichnen den Block b_{source}, über den die Schleife L betreten wird, als ***Quelle*** und jeden Block b_{sink_i}, der eine Kontrollflußkante aus L heraus oder zurück zum Schleifenkopf hat, als ***Senke***. Zusätzlich legen wir einen ausgezeichneten Knoten $b_{supersink}$ an, der als ***Supersenke*** bezeichnet wird. Der Zweck der Supersenke ist, daß wir auf diese Weise einen gemeinsamen Punkt haben, den alle Kontrollflüsse, die die Schleife verlassen, passieren müssen.

b) Für die Berechnung der Kantenmenge E_L filtern wir an jedem Knoten $b \in V_L$ die Menge seiner Nachfolger $b_{succ} \in \delta^+(b)$ und übernehmen nur die Kanten mit $b_{succ} \in V_L$. Dies impliziert, daß wir nur Nachfolger übernehmen, die auf dem WCEP liegen, und daß wir Kanten, die in eine innere Schleife hinein führen, überspringen. Außerdem erzeugen wir für jede Senke b_{sink_i} eine Kante $(b_{sink_i}, b_{supersink})$.

3. Wir definieren $w_L : E_L \to \mathbb{Z}_{\geq 0}$ als eine Gewichtsfunktion auf den Kanten. Wir möchten das Gewicht eines Pfades, das sich aus der Summe seiner Kantengewichte ergibt, so wählen, daß es der WCET dieses Pfades möglichst gut entspricht. Das Gewicht der Kanten zur Supersenke ist 0, sie spielen daher für das Pfadgewicht keine Rolle. Für die anderen Kanten setzen wir das Gewicht wie folgt fest:

$$e = (b_i, b_j) \wedge b_i \neq b_{source} : w_L(e) = WCET_{sum}(b_j) \cdot \frac{WCEC(e)}{WCEC_{sum}(b_j)} \tag{6.1}$$

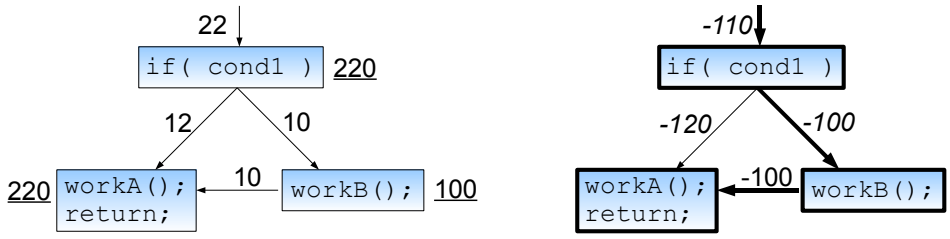
$$e = (b_{source}, b_j) : w_L(e) = WCET_{sum}(b_j) \cdot \frac{WCEC(e)}{WCEC_{sum}(b_j)} + WCET_{sum}(b_{source}) \tag{6.2}$$

Mit der gegebenen Definition von w_L erhalten wir eine möglichst gute Abschätzung der WCET, die auf diesem Pfad anfällt. Jede Kante enthält dabei den Anteil der WCET, der durch Kontrollflüsse über diese Kante am Zielknoten anfällt. Die WCET von b_{source} wird, da es keine Kante gibt, die b_{source} als Zielknoten hat, in jeder Kante die diesen Knoten verlässt gesondert berücksichtigt (Gleichung 6.2).

Die in den Gleichungen verwendeten $WCET_{sum}$-, $WCEC$- und $WCEC_{sum}$-Werte kann man normalerweise direkt aus den `IR_WCETObjects` auslesen, nur bei Knoten $b_{iloop_I} \in V_{iloop}$ für innere Schleifen I müssen diese Werte gesondert berechnet werden. Die $WCET_{sum}(b_{iloop_I})$ ergibt sich dabei als $\sum_{b \in I} WCET_{sum}(b)$, die $WCEC_{sum}(b_{iloop_I})$ als Summe der $WCEC$s der am Schleifenkopf eintreffenden Kanten (außer Rückkanten) und die $WCEC(e)$ mit $e = (b_{iloop_I}, b_{succ})$ kann durch Aufsummierung der $WCEC$s aller der Kanten, die I in Richtung b_{succ} verlassen, ermittelt werden.

4. Wir müssen nun im so erzeugten Graphen einen längsten Pfad von der Quelle zur Supersenke über den Knoten b_{seed} berechnen. Die Berechnung längster Pfade in einem schleifenfreien, gerichteten Graphen G mit Kantengewichten w ist allerdings äquivalent zur Berechnung kürzester Pfade in G mit negierten Kantengewichten $-w$. Dies lässt sich durch einen einfachen Widerspruchsbeweis verifizieren: Die Annahme, daß ein Pfad P_{more} in (G, w) existiert, der länger ist als der ermittelte kürzeste Pfad P_{result}, führt sofort zu dem Schluß, daß P_{more} in $(G, -w)$ ebenfalls ein kürzerer Pfad als P_{result} ist. Dies steht jedoch im Widerspruch zu der Annahme, daß P_{result} der kürzeste Pfad in $(G, -w)$ ist. Die Rückrichtung des Beweises funktioniert analog.

5. Wir negieren daher alle Kantengewichte und können mit dem in der Boost-Bibliothek bereits vorhandenen Bellman-Ford-Algorithmus [Ford & Fulkerson, 1962] nun den kürzesten Weg von der Quelle zu b_{seed} und von b_{seed} zur Supersenke berechnen, der durch die Negation



(a) Codebeispiel mit Kanten-WCECs und Block-WCETs

(b) Selektierter Trace und die Kantengewichte des Longest-Path-Verfahrens

Abbildung 6.7.: Beispiel für die Anwendung des Longest-Path-Algorithmus

der Kantengewichte dem längsten Pfad entspricht. Der etwas bekanntere Algorithmus von Dijkstra ist hier nicht geeignet, da er keine negativen Kantengewichte erlaubt, ebenso ist es nicht möglich, einfach den kürzesten Pfad von der Quelle zur Supersenke zu berechnen, da wir erzwingen wollen, daß b_{seed} auf dem Pfad liegt.

6. Ausgehend von b_{seed} verlängern wir den Trace schrittweise um jeweils einen Block abwechselnd am Anfang und am Ende. Erst in diesem Schritt werden die `IR_SuperblockInfo`-Annotationen erzeugt und mit Inhalt gefüllt. Wir tragen dabei pro Verlängerungsschritt die Traceinformation in das `IR_SuperblockInfo`-Objekt des neu angehängten Blocks ein. Falls wir beim Aufbau des Traces auf einen Knoten $b_{iloop_I} \in V_{iloop}$ stoßen, so wird dieser Knoten *nicht* in den Trace mit aufgenommen, sondern übersprungen. Der Trace-Nachfolger oder Trace-Vorgänger ist dann der Knoten nach oder vor der inneren Schleife I, so daß diese nur implizit Teil des Traces ist. Dies bedingt auch, daß Traces bei unserer Modellierung nicht mit inneren Schleifen starten oder enden können.

   Bei der Verlängerung des Traces berechnen wir in jedem Schritt die erwartete Codegrößenzunahme, die stattfinden würde, wenn dieser Trace in einen Superblock umgeformt würde. Die Grundlage dafür bilden die `IR_CodeSizeObject`-Annotationen. Dazu führen wir eine detaillierte Simulation der Superblockbildung durch, die hier aber nicht näher erläutert werden soll, da die Superblockbildung selbst erst in Abschnitt 6.4 vorgestellt wird. Insgesamt wird der Codegrößenzuwachs bei diesem System durch folgende Regeln beschränkt:

   - Den Faktor, um den die Codegröße des Traces wächst, darf einen benutzerdefinierbaren Grenzwert nicht überschreiten.
   - Analoge benutzerdefinierbare Grenzwerte werden ebenfalls für die optimierte Funktion insgesamt und global für die IR beachtet.
   - Vor der Superblockbildung wird eine Berechnung des verbleibenden freien Programmspeichers vorgenommen (nach der initialen WCET-Berechnung über die LLIR auslesbar). Die erwarteten Codezuwächse werden nur genehmigt, solange mindestens 5% des Gesamtspeichers frei bleiben. Dies stellt sicher, daß der generierte Code noch in den Speicher passt. Aufgrund des relativ großen Speicher des TC1796 tritt dieser Fall in unseren Benchmarks aber nicht auf.

   Das Tracewachstum stoppt in beiden Verlängerungsrichtungen unabhängig voneinander, sobald die Quelle oder eine Senke erreicht wurde, sobald die Codegrößenzuwächse auf dieser Seite nicht mehr genehmigt werden oder sobald man auf einen Knoten stößt, der bereits zu einem anderen Trace gehört.

   Die Codewachstumsvorraussage ermöglicht es uns ebenfalls, eine neue Heuristik für die Be-

stimmung der Richtung des Tracewachstums zu erstellen. In unserer Traceselektion ist es nun auch möglich, den Trace in die Richtung des geringsten Codewachstums zu verlängern, während in bisherigen Publikationen der Trace stets abwechselnd am Anfang und am Ende verlängert wurde.

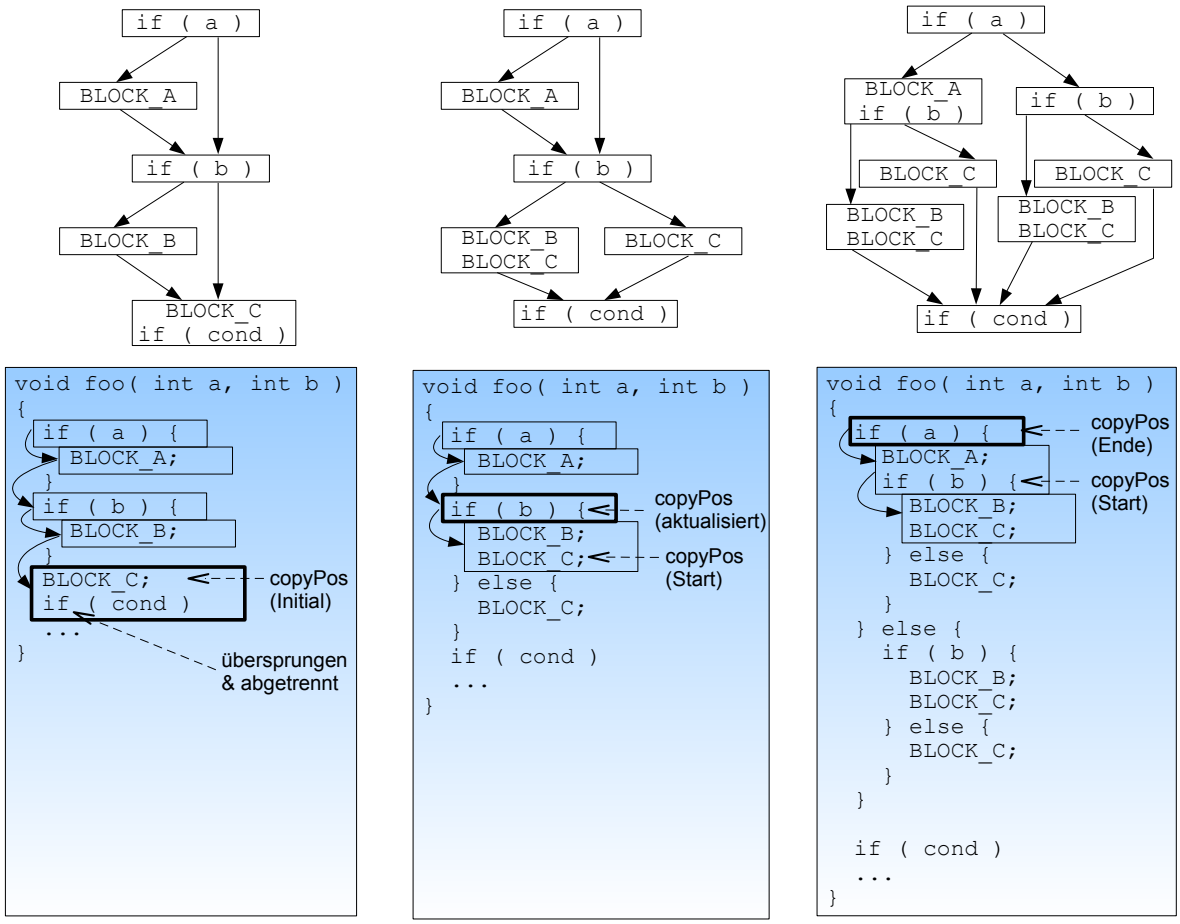
Abbildung 6.7 zeigt die Berechnung der Kantengewichte und die Selektion des längsten Pfades in dem aus Abbildung 6.6 bekannten Beispiel. In Abbildung 6.7(a) sind die Block-WCETs wieder zur Abgrenzung von den Kanten-WCECs unterstrichen dargestellt.

6.4. Superblockerzeugung

Nachdem der gewählte Trace in der IR selektiert wurde, kann die Superblockbildung ihn über die erzeugten `IR_SuperblockInfo`-Annotationen auslesen und zu einem Superblock umformen. Wie bereits in Abschnitt 4.3 dargestellt, unterscheidet sich das verwendete Verfahren jedoch erheblich von dem für die Low-Level-Superblöcke. Wir durchlaufen für die Superblockbildung den Trace vom Ende bis hinauf zum Anfang und eliminieren dabei die vorhandenen Einsprungpunkte, indem wir den bisher gesehenen Trace-Teil in alle Vorgängerzweige kopieren, die nicht auf dem Trace liegen, und ihn danach in den Zweig des Trace-Vorgängers verschieben. Das Vorgehen ist in Algorithmus 6.1 skizziert. Wichtig ist hierbei, daß wir die Anweisungen, die wir verschieben oder kopieren, identifizieren können, indem wir eine Variable `copyPos` vorhalten, die anfangs auf das *Ende* des zu kopierenden oder zu verschiebenden Bereichs zeigt (Zeile 3). Im folgenden wollen wir den Algorithmus anhand des Beispiels aus Abbildung 6.8(a) erläutern. In dieser und den folgenden Abbildungen benutzen wir `BLOCK_XY` als Bezeichnung für einen zusammenhängenden, verzweigungsfreien Abschnitt von Anweisungen, um die Beispiele nicht mit unnötig vielen Details zu überfrachten. Der Block, der der Variable `curBlock` aus Algorithmus 6.1 entspricht, ist fett markiert und die jeweilige `copyPos` ist explizit angegeben. Da die Traceselektion nur ganze Basisblöcke selektieren kann, und das `if ( cond )` Teil des letzten Basisblocks ist, ist die Semantik der Traces so definiert, daß solche `if`s am Ende des Traces nicht Teil des Superblocks werden, sondern nur die restlichen Anweisungen im Block. Falls dies nicht so wäre, wäre es unmöglich anzugeben, daß man nur die Anweisungen vor dem `if` in den Superblock übernehmen möchte, nicht jedoch das komplette `if`, da dies einen erheblich höheren Codegrößenzuwachs verursachen kann. Daher wird das `if ( cond )` in Zeile 6-10 des Algorithmus durch Neusetzen von `copyPos` übersprungen. Danach beginnt das Durchlaufen des Traces (Zeile 14-15) und der Algorithmus stellt schon beim ersten Block fest, daß dort ein Einsprungpunkt in den Trace vorliegt, und versucht, zuerst einige Transformationen durchzuführen, die das Ausmaß der Codevergrößerung beschränken (Zeile 30-37 des Algorithmus). Diese sind in diesem Beispiel jedoch nicht anwendbar; wir werden in den Abschnitten 6.4.2 und 6.4.3 detaillierter auf diese Transformationen eingehen. Der Algorithmus fährt dann in Zeile 39-45 fort und eliminiert hier die eingehenden Kontrollflußkanten. Wir kopieren dazu vom ersten Statement aus `curBlock` (Anfang von `BLOCK_C`) bis `copyPos` (Ende von `BLOCK_C`) alle Anweisungen in den Nicht-Trace-Vorgänger (`else`-Zweig von `if ( b )`) und verschieben die Original-Statements in den Zweig des Trace-Vorgängers (`then`-Zweig von `if ( b )`). Die kopierten Statements gehören danach nicht mehr zum Trace, während die verschobenen Anweisungen auf dem Trace bleiben. Auf diese Art und Weise ist beim Verlassen eines `if`s oder `switch`s, im folgenden zusammengefasst auch als ***Selektionsanweisungen*** bezeichnet, sichergestellt, daß sich der komplette Rest des Traces vom aktuellen Block bis zum Trace-Ende *innerhalb* einer der Verzweigungen befindet. Nach der Elimination der Einsprungkanten behalten wir `curBlock` und `copyPos` bei, um im neuen Compound evtl. vorhandene weitere Einsprungkanten eliminieren zu können. Dies stellt die initiale Situation in Abbildung 6.8(b) dar. Dort ist am dann aktuellen Block `BLOCK_B;BLOCK_C` keine Einsprungkante mehr vorhanden, so daß der Algorithmus in Zeile 51 zum nächsten Trace-Block `if ( b )` wechselt. An dieser Stelle verlassen wir eine Selektionsanweisung und setzen daher diese Anweisung als neue `copyPos` fest, da wir durch das Kopieren der Anweisung auch die Verzweigun-

```
1  void buildSuperblock( Trace t )
2  {
3    IR_Stmt *copyPos = lastStmtOf( endNode( t ) );
4
5    // Überspringe do-while Schleifen oder Ifs/Switches am Ende des Trace
6    if ( isDoWhile( copyPos ) || isConditional( copyPos ) ) {
7      copyPos = precedingStmt( copyPos );
8
9      // Übersprungene Ifs/Switches von Ende des Traces entfernen
10     removeFromTrace( t, endNode( t ) );
11   }
12
13   // Iteriere über den Trace, vom Ende bis zum Anfang
14   IR_BasicBlock *curBlock = endNode( t );
15   while ( curBlock != startNode( t ) ) {
16
17     if ( isConditional( curBlock ) ) {
18       // Der Rest des Traces befindet sich nun innerhalb des If/Switch
19       copyPos = lastStmtOf( curBlock );
20     }
21
22     if ( isPrecededByInnerLoop( curBlock ) ) {
23       // Überspringe die innere Schleife
24       curBlock = tracePredecessorBlock( curBlock );
25     }
26
27     // Eliminiere Einsprungpunkte im aktuellen Block (falls vorhanden)
28     if ( |δ⁻(curBlock)| > 1 ) {
29
30       // siehe Abschnitt 6.4.2
31       if ( causedBySwitch ) {
32         convertSwitch();
33       }
34       // siehe Abschnitt 6.4.3
35       if ( causedByElseIfNest ) {
36         refoldElseIf();
37       }
38
39       for ( IR_BasicBlock *predBlock : δ⁻(curBlock) ) {
40         if ( predBlock == tracePredecessorBlock( curBlock ) ) {
41           moveStmts( firstStmtOf( curBlock ), copyPos, predBlock, ... );
42         } else {
43           copyStmts( firstStmtOf( curBlock ), copyPos, predBlock, ... );
44         }
45       }
46       // 'curBlock' und 'copyPos' bleiben gleich, um im neuen Compound
47       // weitere Einsprungpunkte eliminieren zu können
48     } else {
49
50       // Wechsle zum vorherigen Trace-Block
51       curBlock = tracePredecessorBlock( curBlock );
52     }
53   }
54 }
```

Algorithmus 6.1: Algorithmus zur High-Level-Superblockbildung



(a) Originalcode mit Trace (b) 1. Einsprungpunkt eliminert (c) 2. Einsprungpunkt eliminert

Abbildung 6.8.: Verfolgung des zu kopierenden Bereiches bei der High-Level-Superblockbildung

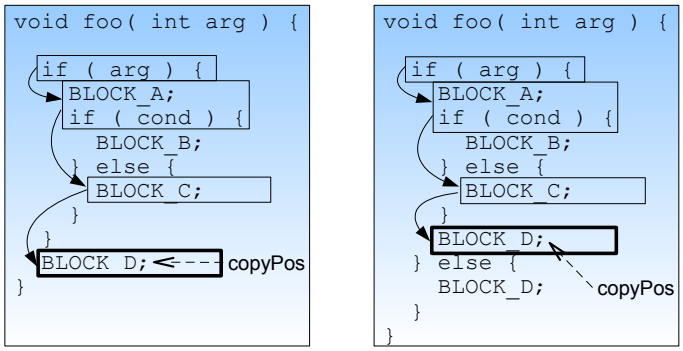
gen (für ein `if` also den `then`- und `else`-Teil) automatisch mit kopieren. Genau dies wird beim Übergang von Abbildung 6.8(b) nach 6.8(c) dann auch ausgeführt. Dort wird die noch verbleibende Einsprungkante eliminiert, indem wir das zum `if ( b )` gehörige `IR_IfElseStmt` kopieren bzw. verschieben. Die Superblockbildung endet in Abbildung 6.8(c) mit dem Erreichen des Trace-Kopfes `if ( a )`.

Im nächsten Abschnitt werden wir die Funktionen `moveStmts` (Zeile 41) bzw. `copyStmts` (Zeile 43) näher untersuchen, da diese nicht immer so problemlos umsetzbar sind wie in dem einfachen Beispiel aus Abbildung 6.8.

Die Aufrufe der Funktionen `convertSwitch` und `refoldElseIf` werden in Abschnitt 6.4.2 und 6.4.3 genauer besprochen, sie dienen der Minimierung des erzielten Codegrößenzuwachses.

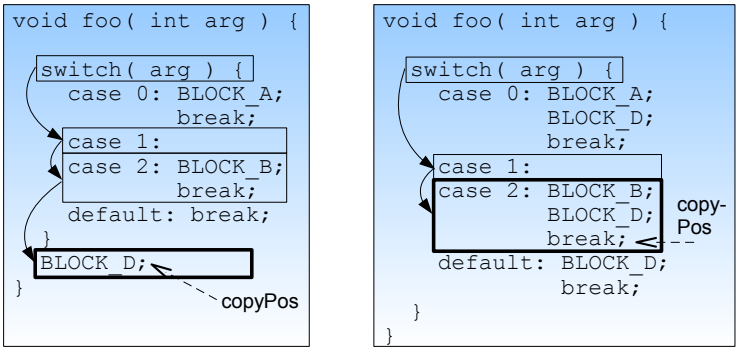
6.4.1. Elimination von Einsprungpunkten

Da wir Rücksprungkanten von inneren Schleifen auf dem Trace nicht als Einsprungkanten betrachten, sondern die gesamte innere Schleife als elementare Anweisung behandeln, müssen wir beim Eliminieren der Einsprungpunkte nur zwischen drei verschiedenen Szenarien unterscheiden. Diese werden wir im folgenden näher beschreiben.



(a) Originalcode mit Trace (b) 1. Einsprungpunkt eliminert

Abbildung 6.9.: Einsprungpunkte durch `if`-Anweisungen

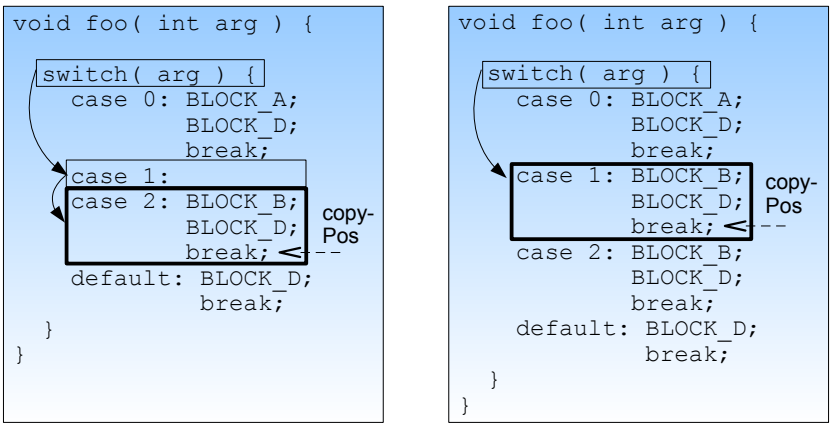


(a) Originalcode mit Trace (b) 1. Einsprungpunkt eliminert

Abbildung 6.10.: Einsprungpunkte durch `switch`-Anweisungen

Einsprungpunkte durch `if`-Anweisungen

Das erste Szenario wurde bereits in Abbildung 6.8 dargestellt und seine Behandlung ist relativ einfach. Das `if` verursacht exakt *eine* eingehende Nicht-Trace-Kante und *eine* Trace-Kante. Wir rufen daher `copyStmts` und `moveStmts` jeweils einmal auf und eliminieren dadurch die eingehende Nicht-Trace-Kante. Falls der aktuelle Block noch weitere eingehende Kanten hatte (wie es z.B. in Abbildung 6.9(a) bei `BLOCK_D` der Fall ist), so eliminieren wir zuerst nur die Nicht-Trace-Kante des `ifs`, das sich direkt vor dem aktuellen Block befindet. Die weiteren Einsprungkanten werden in den nachfolgenden Schritten eliminiert. Außerdem ist in Abbildung 6.9(b) ebenfalls zu erkennen, daß wir für diesen Schritt Objekte vom Typ `IR_IfStmt` zu `IR_IfElseStmt`s umwandeln müssen, wobei alle evtl. an diese Statements angehängten Annotationen mit an das neue Statement übertragen werden müssen. Eine Besonderheit, die wir außerdem beachten müssen, ist, daß wir bei `if`-Statements, bei denen einer der Zweige durch ein `break`, `continue` oder `return` beendet wird (die also keine Einsprungkante verursachen), den Trace-Code trotzdem in den Pfad verschieben müssen, der auf dem Trace liegt. Dies ist nötig, da wir bei Verlassen des `if`s davon ausgehen werden, daß sich der vollständige Rest des Traces *innerhalb* des `if`s befindet und durch die Verschiebung stellen wir genau dies sicher.



(a) Originalcode mit Trace (b) 1. Einsprungpunkt eliminert

Abbildung 6.11.: Einsprungpunkte durch `case`-Anweisungen

Einsprungpunkte durch `switch`-Anweisungen

Die Behandlung von Einsprungpunkten, die durch `switch`-Anweisungen verursacht werden, ist etwas komplexer. Wir können durch den Prepass davon ausgehen, daß jede eingehende Kontrollflußkante aus dem switch heraus durch ein `break` verursacht wird und ebenso davon, daß das `switch` ein `default` enthält, es also keine Möglichkeit gibt, das `switch` zu überspringen. Daher können wir uns darauf beschränken, den zu kopierenden Code-Teil vor jedes `break` im `switch` zu kopieren, oder ihn im Falle des Trace-Vorgänger-Knotens dorthin zu verschieben. Ein Beispiel hierfür ist in Abbildung 6.10 gegeben. Allerdings verschieben wir hier Code in die Mitte eines Basisblocks, wie z.B. in Abbildung 6.10(b) illustriert. Daher kann es dazu kommen, daß der betroffene Basisblock gespalten wird, nämlich genau dann, wenn wir eine innere Schleife oder eine Selektionsanweisung mit kopieren. Um diesen Fall müssen sich die `copyStmts` bzw. `moveStmts` Methoden durch Anpassung der Annotationen (die ja pro Block vorhanden sein müssen) kümmern. Diese Methoden werden wir weiter unten genauer beschreiben.

Einsprungpunkte durch `case/default`-Fall-Through-Pfade

Das dritte und letzte mögliche Szenario ist eine Einsprungkante durch einen ***Fall-Through-Path*** in einem `switch` wie er in Abbildung 6.11(a) zu sehen ist. In diesem Fall müssen wir zuerst feststellen, bis zu welcher Stelle im `switch`-Compound wir den Code des aktuellen `case`- oder `default`-Blocks kopieren müssen (im folgenden werden wir nur noch von `case`s reden, das `default` ist dabei jeweils mit eingeschlossen). Falls im Compound ein `break` direkt auf das `case`-Statement folgt (wie in Abbildung 6.11(a)), so enden die Kopieroperationen an diesem `break`. Im Allgemeinen können diese `break`s aber auch innerhalb von `if`s sitzen oder aber der Kontrollfluß durch ein `return` beendet werden. Deshalb bestimmen wir das Statement, bis zu dem wir kopieren müssen, indem wir ausgehend vom aktuellen `case`-Statement s_{start} die Statements des `switch`-Compounds durchlaufen und stoppen bei Statement s_{stop}, sobald das Compound-Ende errreicht ist oder wir auf ein Statement stoßen, das vom letztgesehenen Statement aus nicht erreichbar ist. Im Beispiel stoppen wir daher beim `break` nach `BLOCK_D`, da es keinen möglichen Kontrollfluß von `break` nach `default` gibt, der innerhalb des `switch`es verläuft. Danach rufen wir `copyStmts`(s_{start}, s_{stop}, *preceding_case*) auf. `copyStmts` erhält hierbei jedoch noch einen weiteren Parameter, der angibt, ob der Fall-Through-Path auf dem Trace liegt. Falls dies so ist, verschiebt `copyStmts` die Original-Statements in das vorhergehende `case`, von dem der Fall-Through-Path ausgeht, und fügt Kopien der Original-Statements im aktuellen `case` ein. Abbildung 6.11 zeigt einen solchen Fall. Ansonsten, falls der

Trace also nicht durch den Fall-Through-Path verläuft, werden wie in den bekannten Fällen nur Kopien erzeugt und an der Zielposition, hier also im vorhergehenden `case`, eingefügt. Da wir immer einen Codeabschnitt kopieren, der über `break` oder `return` Anweisungen beendet wird, ist der Fall-Through-Path nach der Kopieroperation eliminiert.

Hilfsfunktion `copyStmts`

Die in den letzten Abschnitten mehrfach erwähnte Funktion zum Kopieren der Trace-Statements, `copyStmts`, wird im folgenden vorgestellt. Aufgabe der Funktion ist es, Kopien der gegebenen Trace-Statements zu erzeugen und an die passende Stelle zu verschieben. Die Parameter der Funktion sind:

1. `IR_Stmt *copyStart`
   Der Beginn des zu kopierenden Bereiches
2. `IR_Stmt *copyEnd`
   Das Ende des zu kopierenden Bereiches (wird mitkopiert)
3. (`IR_CompoundStmt*`, `iterator`) `insertPosition`
   Ein Zielcompound und die Position darin, an der die kopierten Statements eingefügt werden sollen
4. `bool moveCopiedStmts`
   Die Angabe, ob die kopierten, oder aber die originalen Statements an die `insertPosition` verschoben werden sollen (siehe Abschnitt über die `case`-Einsprungkanten).

Der Kopierprozeß ist dabei relativ aufwendig, weil wir bei den kopierten Statements alle Annotationen konsistent halten müssen. Im folgenden werden wir erläutern, was dafür im Einzelnen nötig ist:

1. **Zielcompound-Symbole umbenennen**
   Falls der zu kopierende Code Symbole definiert und das Zielcompound gleichnamige Symbole neu deklariert, so muß jeweils eines der gleichnamigen Symbole umbenannt werden, damit die Semantik erhalten bleibt. Wir benennen hierbei die gleichnamigen Symbole im Zielcompound durch das Anhängen eines Präfix "_sbform[nr]_" um, wobei [nr] durch eine für jedes Symbol eindeutige ID ersetzt wird.

2. **Statische Variablen extrahieren**
   Falls im zu kopierenden Code statische Variablen deklariert werden, z.B. in inneren Schleifen oder der evtl. mit kopierten Selektionsanweisung, so müssen diese Variablen in einem gemeinsamen, äußeren Compound deklariert werden, damit weiterhin eine einzelne und nicht mehrere unabhängige statische Variablen gleichen Namens existieren. Wir benennen diese statischen Variablen daher zur Vermeidung von Namenskonfliken ebenfalls wie oben um und verschieben ihre Deklaration in das Compound, das `copyStart` und `copyEnd` enthält.

3. **Kopiere die Statements**
   Wir kopieren hierbei angefangen bei `copyStart` bis `copyEnd` jedes Statement $s_1, ..., s_n$ einzeln und verschieben es direkt nach `insertPosition`. Dabei müssen wir allerdings `case`- und `default`-Statements überspringen, da diese nicht mitkopiert werden (dieser Fall tritt nur bei der Behandlung von Einsprungkanten aus Fall-Through-Pfaden auf):

   a) Aktuelles Statement s_i ist `case` oder `default`

      i. $s_i = copyStart \vee s_{i-1}$ ist innere Schleife
         $\Rightarrow$ Speichere die Annotationen des `case`/`default` in einem Annotationsbackup

      ii. $s_i \neq copyStart \wedge s_{i-1}$ ist keine innere Schleife

A. Es sind bereits Statements verschoben worden
⇒ Verschmilz die Annotationen des letzten verschobenen Blocks mit denen des `case`/`default` (Funktion `mergeAnnotations`, s.u.)

B. Sonst: Es wurden bereits andere `case`s/`default`s überprungen
⇒ Verschmilz deren Annotationsbackup mit den Annotationen des `case`/`default` (Funktion `mergeAnnotations`, s.u.)

b) Aktuelles Statement s_i ist *kein* `case` oder `default`

- Kopiere s_i.
- Übertrage Kopien der Flowfacts von s_i zum neuen Statement. Der Flowfactmanager passt dabei automatisch die betroffenen Flowfacts an. Hierbei müssen jedoch nie Schleifengrenzen verändert werden, da die ursprünglichen Grenzen auch für den Kontrollfluß über die `insertPosition` gültig sein mussten.
- Hänge Kopien der `IR_WCETObject-` und `IR_CodeSizeObject`-Annotationen von s_i an das neue Statement an.
- Verschiebe die Kopie (falls `moveCopiedStmts` = `true`) oder das Original (sonst) vor die `insertPosition`
- Falls Annotationsbackup vorhanden: Zum verschobenen Statement hinzufügen

4. **Annotationen zurücksetzen**
Die `IR_WCETObject-` und `IR_CodeSizeObject`-Annotationen, die an die neu erzeugten Kopien angehängt wurden, werden zurückgesetzt. Dabei werden nur die Iteration-WCET, die Infeasibility-Information, die Codegröße und der Referenzblock (s. Abschnitt 3.4.2) übernommen, da wir im folgenden bei der Neuberechnung des WCEP und der nächsten Traceselektion nur diese Werte erneut brauchen werden. Alle anderen Annotations-Werte, wie z.B. die Kanten-WCECs, werden gelöscht, da sie durch die Verschiebung möglicherweise nicht mehr korrekt sind. Hierbei ist zu beachten, daß die Annotationen bei der Superblockbildung bereits in der Statement-basierten Form vorliegen, daher ist der Referenzblock hier in Form eines Referenz-Statements, also eines Pointers auf das erste Statement des Referenzblock, gespeichert. Da wir dieses Statement evtl. auch kopiert haben (z.B. wenn eine komplette innere Schleife kopiert wurde), müssen wir alle Referenz-Statements in den Kopien überprüfen und gegebenenfalls mittels einer Zuordnungstabelle "Originalstatement → Kopie" auf die kopierten Statements umlenken. Diese Zuordnungstabelle wird während des Kopierens angelegt.

5. **Behandlung abgetrennter `breaks`**
Falls wir Code vor ein `break`-Statement verschieben, wie es bei der Elimination von Einsprungpunkten durch `switch`-Statements vorkommt, so kann es dazu kommen, daß hierdurch das `break` von seinem ursprünglichen Basisblock getrennt wird. Diese Änderung müssen wir an die `IR_WCETObject-` und `IR_CodeSizeObject`-Annotationen weiterreichen. Dabei gibt es verschiedene Aspekte zu beachten.

- Falls das `break` Teil eines Basisblocks mit ≥ 2 Statements war, entferne die WCET und Codegröße des `break`s aus diesem Block. Die Werte werden dabei über die Schätzverfahren aus Abschnitt 5.2 berechnet.
- Falls das `break` *jetzt* Teil eines neuen Blocks mit ≥ 2 Statements ist, addiere die `break`-WCET und -Codegröße zu diesem Block hinzu.
- Falls das `break` jetzt einen eigenen Block darstellt (möglich durch Kopieren eines Selektionsstatements vor das `break`), so erzeuge für diesen neuen Block neue `IR_WCETObject-`

und `IR_CodeSizeObject`-Annotationen. Die WCET- und Codegrößen-Werte werden dabei wieder geschätzt.

- Falls das `break` durch die Transformation unausführbar geworden ist (z.B. durch Kopieren eines `return`-Statements vor das `break`), dann lösche das `break`.

6. **Kopf-Annotationen verschmelzen**
   Bisher haben wir nur die Annotationen *im* kopierten Bereich (Behandlung von nicht mitkopierten `case`-Statements) und die Annotationen *hinter* dem kopierten Bereich (Behandlung abgetrennter `break`s) aktualisiert. Es kann jedoch auch *davor* zu Inkonsistenzen kommen:

   Wenn wir durch das Kopieren des ersten Basisblocks im kopierten Bereich, also dem Block von `copyStart`, zwei Basisblöcke miteinander verschmolzen haben (dies ist z.B. in Abbildung 6.10(b) der Fall), so müssen wir auch ihre WCET- und Codegrößen-Annotationen miteinander verschmelzen. Dafür steht eine Funktion `mergeAnnotations( IR_Stmt a, IR_Stmt b )` zur Verfügung, die diese Aufgabe übernimmt. Die Funktion löscht eine evtl. vorhandene Referenzbeziehung zwischen den beiden Blöcken oder addiert die Iteration-WCET und die Codegrößen der Blöcke, falls sie nicht in einer Referenzbeziehung stehen. Die Ergebnisse werden in Statement `a` eingetragen, während die Annotationen von Statement `b` entfernt werden. In unserer Anwendung ist also `a` = `copyStart` und `b` = letzter Block vor `insertPosition`.

Hilfsfunktion `moveStmts`

Die `moveStmts`-Funktion funktioniert in weiten Teilen exakt wie die `copyStmts`-Funktion. Der Unterschied ist, daß wir diese Funktion nutzen, um Statements in `if`- oder `switch`-Zweige zu verschieben, die *auf* dem Trace liegen, während wir `copyStmts` benutzen, um Statements auf Zweige zu verschieben, die *nicht* auf dem Trace liegen. Die Parameter von `moveStmts` sind:

1. `IR_Stmt* moveStart`
   Der Beginn des zu verschiebenden Bereiches
2. `IR_Stmt* moveEnd`
   Das Ende des zu verschiebenden Bereiches (wird mitverschoben)
3. (`IR_CompoundStmt*`, `iterator`) `insertPosition`
   Ein Zielcompound und die Position darin, an der die Statements eingefügt werden sollen

Die einzelnen Punkte zur Durchführung werden hier noch einmal kurz beschrieben:

1. **Zielcompound-Symbole umbenennen**
   Dieser Schritt funktioniert genauso wie bei `copyStmts`.

2. **Statische Variablen extrahieren**
   Auch dieser Schritt funktioniert genauso wie bei `copyStmts`.

3. **Kopiere die Statements**
   Hier werden in `moveStmts` natürlich die Statements verschoben und nicht kopiert. Dadurch bleiben auch alle Annotationen erhalten, so daß wir diese nicht neu erzeugen müssen. Außerdem entfällt die Sonderbehandlung für `case`-Statements, da wir diese nie verschieben.

4. **Annotationen zurücksetzen**
   Die WCET- und Codegrößen-Informationen werden auch in den verschobenen Statements zurückgesetzt. Die Superblock-Informationen müssen nicht verändert werden, sie bleiben unverändert.

5. **Behandlung abgetrennter `breaks`**
   Hier muß zusätzlich der Superblock-Pfad repariert werden, das `break` steht danach in jedem Fall weiter "hinten" im Superblock als vor der Verschiebe-Operation.

6. **Kopf-Annotationen verschmelzen**
   Hier müssen auch die `IR_SuperblockInfo`-Objekte der beiden Blöcke miteinander verschmolzen werden. Der jetzt nicht mehr einzeln vorhandene Block (der Block von `moveStart`) wird dabei aus dem Superblock entfernt. Der Superblock verläuft danach also direkt von "letzter Block vor `insertPos`" nach "Trace-Nachfolger von `moveStart`".

7. **Sonderbehandlung beim Verschieben halber Blöcke**
   Dieser Schritt fällt ausschließlich bei `moveStmts` an. Wie anfangs erwähnt, werden Selektionsanweisungen, die das letzte Statement des Traces sind, von der Superblockbildung ausgeschlossen. Falls diese Statements nicht allein in ihrem Basisblock sind, werden die anderen Statements jedoch in die Superblockbildung miteinbezogen, werden also von `moveStmts` aus ihrem alten Block entfernt. Zurück bleibt dabei die Selektionsanweisung, die danach nicht mehr Teil des Superblocks ist. In diesem letzten Schritt von `moveStmts` wird also überprüft, ob wir durch die Verschiebung einen Block zerteilt haben und die Iteration-WCET und Codegrößen-Werte der beiden Blockteile werden aktualisiert. Dazu wird das Schätzverfahren nach Abschnitt 5.2 benutzt.

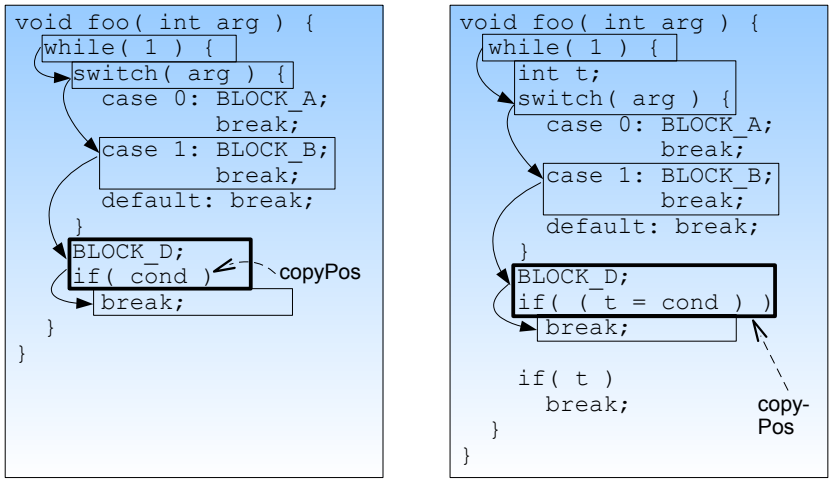
Behandlung von nicht verschiebbaren `switch breaks`

Die bisher vorgestellten Funktionen gehen davon aus, daß es legal ist, Anweisungen, die ursprünglich hinter einer Verzweigung standen, in die Verzweigung hinein zu kopieren. Dies ist auch fast immer der Fall, allerdings haben wir schon in der Beschreibung der `copyStmts`-Funktion gesehen, daß es Ausnahmen gibt, auf die diese Annahme nicht zutrifft, wie z.B. Code, der statische Variablen deklariert. In diesem Fall mußten wir zusätzliche Maßnahmen ergreifen, damit der Code verschiebbar wird, nämlich die statische Deklaration in ein gemeinsames, äußeres Compound verschieben. Es gibt noch einen weiteren Fall, in dem eine Verschiebung der Anweisungen nicht ohne weiteres möglich ist.

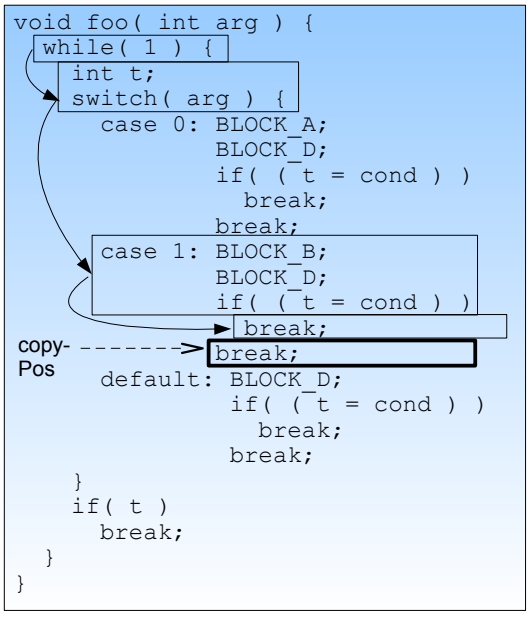
Die `break`-Anweisungen sind die einzigen Anweisungen, deren Semantik sich durch das Verschieben in eine Verzweigung, genauer gesagt in ein `switch`, ändert: Wenn das `break` vorher eine umgebende Schleife oder ein umgebendes `switch` abgebrochen hat, so wird es nach der Verschiebung zu einem Abbruch des *inneren* `switch`-Statements führen. Ein Beispiel für eine solche Situation ist in Abbildung 6.12(a) dargestellt, wo das `break` am Ende des Traces nicht in das `switch` verschoben werden kann. In solchen Fällen schließen wir das unverschiebbare `break` vom Superblock aus. Es verbleibt dabei nach der Superblockbildung in seinem jeweiligen Bedingungskontext hinter dem aktuell bearbeiteten `switch`-Statement. Im Detail gehen wir folgendermaßen vor:

1. Falls das `break` im aktuellen Block steht (`copyPos` = `break`), so überspringen wir dieses `break` beim Verschieben / Kopieren der Statements und lassen es auf diese Weise hinter dem Superblock zurück

2. Sonst besteht nur noch die Möglichkeit, daß das `break` in einer Schachtelung von `if`-Statements steht. Wenn in dieser Schachtelung andere Statements wie z.B. Schleifen oder andere `switches` vorkämen, dann wäre das betroffene `break` nicht unverschiebbar. Ein Beispiel für diese Situation ist in Abbildung 6.12(a) gegeben. Wir erzeugen daher eine Kopie der `if`-Hierarchie für jedes unverschiebbare `break` und fügen diese Kopie hinter dem Superblock ein (s. Abbildung 6.12(b)). Die Resultate der verwendeten Bedingungen müssen dabei im Allgemeinen zwischengespeichert werden, wenn wir nicht sicherstellen können, daß sie keine Variablen beschreiben oder andere Nebeneffekte haben. Danach wird eine normale Superblockbildung

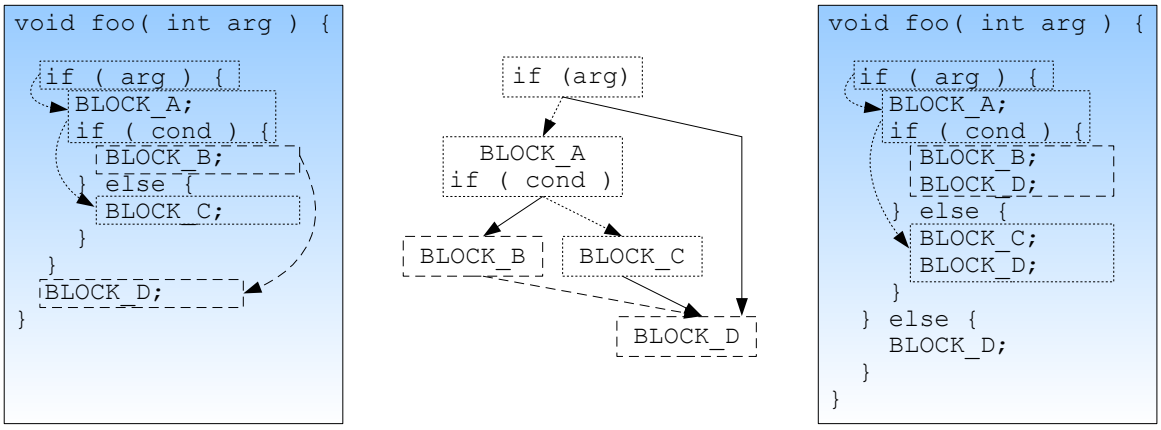


(a) Originalcode mit Trace (b) Unverschiebbares `break` extrahiert



(c) Superblock mit `break`-Anhang

Abbildung 6.12.: Behandlung nicht verschiebbarer `break`-Anweisungen



(a) Originalcode (b) CFG des Originalcodes (c) Nach der Superblockbildung

Abbildung 6.13.: Beispiel zur Stabilität der Superblockbildung

durchgeführt, wobei das ursprünglich unverschiebbare `break` in das vorausgehende `switch` verschoben wird (s. Abbildung 6.12(c)). Wenn es nun innerhalb des inneren `switch` erreicht wird, springt der Kontrollfluß sofort zu den kopierten `if`-Hierarchien und erreicht dort die Kopie des jeweiligen `break`s, so daß auch das umgebende Statement verlassen wird, so wie im ursprünglichen Kontrollfluß.

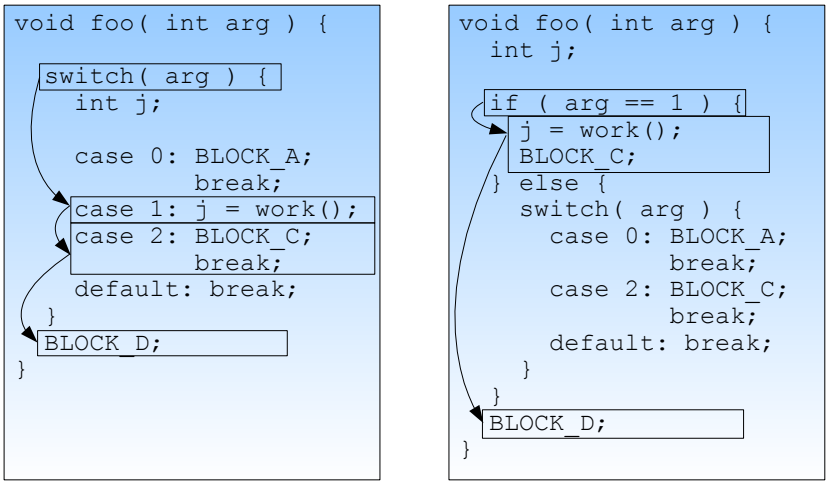
Durch diese Transformationen ist sichergestellt, daß die Semantik des kompilierten Programms auch bei nicht verschiebbaren `break`-Statements erhalten bleibt.

Stabilität der Superblockbildung

In diesem Abschnitt wollen wir abschließend erläutern, warum es nicht möglich ist, daß durch die Neuerzeugung eines Superblocks $Y = (b_{y_1}, ..., b_{y_n})$ ein bereits bestehender Superblock $X = (b_{x_1}, ..., b_{x_n})$ zerstört wird. Dies wäre nur dann möglich, wenn wir während der Superblockbildung ein Stück Code aus Y in X hinein kopieren, so daß eine neue Einsprungkante in X entsteht. Wir kopieren aber nur dann Code nach X, wenn es eine Kontrollflußkante (b_s, b_t) mit $b_s \in \{b_{x_1}, ..., b_{x_{n-1}}\}$ und $b_t \in \{b_{y_2}, ..., b_{y_n}\}$ gibt. Eine solche Einsprungkante, ist z.B. in Abbildung 6.13(a) und 6.13(b) mit der Kante von `if(arg)` nach `BLOCK_D` dargestellt. Der fertige Superblock X ist dort gepunktet und der noch unvollständige Superblock Y gestrichelt markiert. Diese problematischen Einsprungkanten können in schleifenfreiem[1] High-Level-Code nur durch ein `if`-Statement ohne `else`-Zweig, ein `switch`-Statement ohne `default`-Zweig oder einen Fall-Through-Pfad in einem `switch`-Statement bewirkt werden. Den Fall des `switch`-Statement ohne `default` können wir ausschließen, da der Prepass erzwingt, daß in jedem `switch` ein `default`-Statement existiert, und die Argumentation zu Fall-Through-Pfaden funktioniert analog zu derjenigen bei `if`s, so daß wir uns im folgenden auf diesen Fall beschränken werden.

In Abbildung 6.13(a) ist der entsprechende Code zur Situation bei einer Einsprungkante durch ein `if`-Statement dargestellt. Man erkennt, daß aufgrund der syntaktischen Eigenschaften der High-Level-Darstellung die problematischen Einsprungkanten von allen `if`-Statements ausgehen, die auf X liegen und keinen `else`-Zweig besitzen. Da die High-Level-Superblockbildung, wie in Abbildung 6.13(c) zu sehen ist, jedoch immer nur Code am *Ende* der jeweiligen Compounds einfügt, wird auch X höchstens am *Ende* verlängert, was die Superblockeigenschaft nicht zerstört.

[1] Unser Code ist zwar nicht schleifenfrei, aber da wir innere Schleifen als elementare Anweisungen behandeln besitzt er effektiv dieselben Eigenschaften.



(a) Originalcode mit Trace (b) `switch` wurde konvertiert

Abbildung 6.14.: Beispiel für `switch`-Konvertierung

Zusammenfassung

Die grundlegenden Algorithmen, die wir zur Erzeugung der Superblöcke benutzen, sind damit vollständig vorgestellt. Im folgenden werden wir allerdings noch zwei Transformationen kennen lernen, die in manchen Situationen das Ausmaß der Codegrößenzunahme beschränken und die WCET verbessern können. Im einzelnen sind dies die Extraktion des Worst-Case-Pfades aus einem `switch` und die Neufaltung verschachtelter `if-else`-Statements.

6.4.2. `switch`-Konvertierung

Die `switch`-Konvertierung versucht, bei der Elimination von Einsprungpunkten durch `switch`-Statement, den Trace-Abschnitt, der innerhalb des `switch`es liegt, in ein neu anzulegendes, umgebendes `if` zu verlagern. Ein Beispiel hierfür ist in Abbildung 6.14 dargestellt. Die positiven Effekte dieser Transformation sind das geringere Codegrößenwachstum bei der Superblockbildung (es muß nur noch *eine* statt (#cases − 1) Kopien erzeugt werden) und die mögliche Verkürzung des WCEP (die Verzeigung zum Superblock wird nach der Transformation immer zuerst ausgeführt).

Bei jedem `switch`, daß während der Superblockbildung passiert werden muß, wird versucht die `switch`-Konvertierung anzuwenden (s. Algorithmus 6.1, Zeile 32). Dabei muß jedes Mal neu entschieden werden, ob dies möglich ist. Die Kriterien hierfür sind:

- Der Trace betritt das switch *nicht* über den `default`-Zweig:
  Für diesen Fall könnten wir bei der gewählten Konvertierungsmethode keine sinnvolle Bedingung für das neu anzulegende `if` angeben.
- Das `switch` enthält keine bedingt ausgeführten `break`s:
  Falls bedingt ausgeführte `break`s auf dem Codebereich liegen würden, den wir in das `if` verschieben, so verlieren diese `break`s ihre Bedeutung, da sie nicht mehr im `switch` stehen. Eine Ersetzung durch `goto`s ist in unserem Rahmenwerk auch nicht möglich, da wir `goto`s generell nicht zulassen.
- Es befinden sich noch keine Superblöcke im `switch`, da wir sonst nicht sicherstellen könnten, daß diese durch die `switch`-Konvertierung nicht zerstört würden.

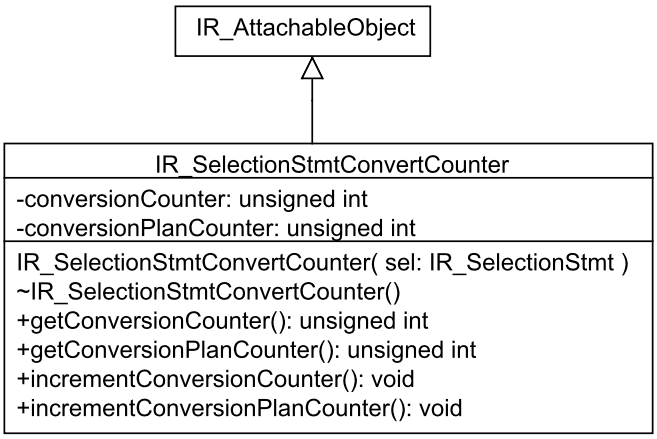


Abbildung 6.15.: Aufbau der `IR_SelectionStmtConvertCounter`-Annotation zum Vermerken geplanter und bereits ausgeführter Konvertierungen von `switch`-Statements.

- Gesamtzahl möglicher Konvertierungen für dieses `switch` wurde noch nicht überschritten:
  Für jedes `switch` auf einem Trace wird schon in der Traceselektion, bei der Überprüfung des Codegrößenwachstums festgestellt, ob es später konvertiert werden soll oder nicht. Über eine Annotation vom Typ `IR_SelectionStmtConvertCounter` (s. Abbildung 6.15), die an jedes `switch` auf einem Trace angehängt wird, wird festgesetzt ob und wie oft dieses `switch` später konvertiert werden darf. Jedes `switch` kann potentiell mehrmals konvertiert werden, wobei in jeder neuen Konvertierung ein anderes `case`-Statement extrahiert wird. So wäre es z.B. möglich, daß im verbleibenden `switch` in Abbildung 6.14(b) erneut ein Trace selektiert wird, bei dessen Superblockbildung eine erneute `switch`-Konvertierung möglich wäre, z.B. durch den `case 2`. Deswegen hält die `IR_SelectionStmtConvertCounter`-Annotation auch fest, wie oft das annotierte `switch` bereits konvertiert wurde, so daß im Verlauf der Superblockbildung später feststellbar ist, ob der gesetzte Grenzwert schon erreicht wurde. In diesem Fall wird keine weitere Konvertierung mehr durchgeführt. Im aktuellen WCC kann der Benutzer angeben, ob bei der Superblockbildung überhaupt eine `switch`-Konvertierung durchgeführt werden soll. Falls ja gilt pro `switch` eine Obergrenze von 1 für die Gesamtzahl der Konverterungen dieses `switch`es, da der Nutzen der Transformation mit zunehmender Anzahl der Konvertierungen abnimmt.

Einen ersten Eindruck von der Funktionsweise der `switch`-Konvertierung liefert bereits Abbildung 6.14. Im folgenden beschreiben wir, wie die Konvertierung im einzelnen abläuft. Die einzige Eingabe für den Algorithmus ist das zu konvertierende `switch`-Statement.

1. **Vorbedingungen klären**
   Vor dem Beginn der Konvertierung ist zu klären ob die erwähnten Vorbedingungen erfüllt sind. Ist dies nicht der Fall, so wird die Konvertierung abgebrochen.

2. **Trace-Ende im `switch` identifizieren**
   Wir ermitteln den Trace-Vorgänger-Block b_{pred} von `curBlock` (s. Algorithmus 6.1, Zeile 14/24/51) auf unserem aktuellen Trace T_{cur}. Dieser Block ist der letzte Trace-Block innerhalb des zu konvertierenden `switch`-Statements. Die Variable `traceEndInSwitch` definieren wir als Pointer auf `lastStmtOf(` b_{pred} `)` oder auf das davor liegende Statement, falls `lastStmtOf(` b_{pred} `)` ein `break` ist. Diese Variable markiert später das Ende der Kopieroperationen. Im Beispiel ist dieses Statement das Ende von `BLOCK_C`. Falls `traceEndInSwitch` ein Statement ist, das nicht im `switch`-Compound liegt, so suchen wir das umgebende Statement

$s_{top} \geq_s$ `traceEndInSwitch`, das im `switch`-Compound liegt, und setzen `traceEndInSwitch` = s_{top}.

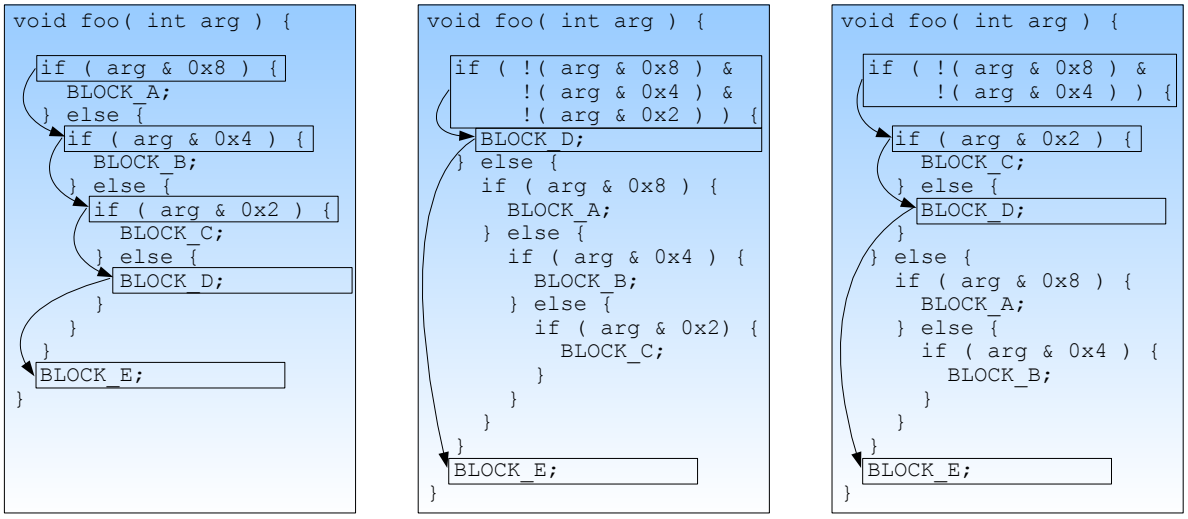
3. **Trace-Start im `switch` identifizieren**
   Passed dazu müssen wir noch die Stelle ermitteln, an der die Kopieroperationen beginnen sollen. Da dies stets ein `case`-Statement ist, bezeichnen wir die Variable als `traceCase`. Da wir die Superblockbildung nur anwenden, wenn alle `switches` ***strukturiert*** sind, d.h. alle ihre `case`-Labels im `switch`-Compound haben, können wir zum Bestimmen der Startposition einfach die Statements des `switch`-Compounds in Richtung Beginn durchlaufen und stoppen, sobald wir auf ein `case`-Label stoßen, das entweder den `switch`-Block als Trace-Vorgänger-Block hat, oder aber gar keinen Trace-Vorgänger hat. In unserem Beispiel aus Abbildung 6.14 würde das bedeuten, daß wir `case 2:` überspringen (der Trace-Vorgänger ist dort `case 1:`) und erst bei `case 1:` halten.

4. **Erzeugung des `if`-Statements**
   Wir erzeugen ein neues `if`-Statement mit der Bedingung `conditionVariable(switch) == nr_of_case(traceCase)`, verschieben vorläufig das komplette `switch` in den `else`-Zweig dieses `if`s und fügen das `if` an der Stelle, an der vorher das `switch` stand ein. Hierbei können verschiedene Annotationen ungültig geworden sein, die wir daher anpassen müssen:

   - Falls das `switch` der Kopf eines Basisblocks war (dort werden die Annotationen gespeichert), so müssen wir für das `if` völlig neue `IR_WCETObject`-, `IR_CodeSizeObject`- und `IR_SuperblockInfo`-Annotationen erzeugen. Die Iteration-WCET- und Codegrößen-Werte werden hierbei mit unserer bekannten Heuristik geschätzt. Der Trace $T_{switch-head}$ auf dem das `switch` lag, wird so aktualisiert, daß das `if` jetzt den Platz des `switch` einnimmt. Hierbei ist zu beachten, daß es keineswegs sicher ist, daß $T_{switch-head} = T_{cur}$ ist. Es kann sein, daß ein anderer Trace sich bis zum Block des `switch` erstreckt und T_{cur} erst innerhalb des `switch` beginnt.
     Außerdem müssen in diesem Fall die Referenz-Statements in allen *anderen* Basisblöcken aktualisiert werden, deren Referenz-Statement das `switch` war. Das neue Referenz-Statement dieser Blöcke ist das neu erzeugte `if`.
   - Falls das `switch` vorher *nicht* der Kopf eines Basisblocks war, so ist es durch unsere Transformation dazu geworden und benötigt daher ebenfalls neue Annotationen. Die Iteration-WCET- und Codegrößen-Werte werden hierbei wieder geschätzt, eine Zugehörigkeit zu einem Superblock ist nicht mehr gegeben, da wir in den Vorbedingungen ausschließen, daß im `switch` bereits andere Traces/Superblöcke existieren.

5. **Verschieben der Trace-Statements**
   Wir nutzen `copyStmts( traceCase, traceEndInSwitch, newIf.getThenPart(), false )` um die originalen Statements in den `then`-Zweig des neuen `if`s zu verschieben und Kopien der Statements im `switch` einzufügen. Dies entspricht exakt dem Vorgehen bei der Eliminierung von Fall-Through-Pfaden in `switches`. Beim Kopieren werden dabei, wie in der Beschreibung der `copyStmts`-Funktion erwähnt, die `case`-Statements nicht mitkopiert. `breaks` können ebenfalls nicht mitkopiert werden, da wir bedingte `breaks` nicht erlauben, und unbedingte `breaks` die Positionierung von `traceEndInSwitch` verändern würden.

6. **Reparatur des Trace-Pfads**
   Nach der Verschiebung muß der Superblock an zwei Stellen repariert werden. Innerhalb des kopierten Bereichs sind die Annotationen durch die Verschiebung intakt geblieben, da es sich immer noch um dieselben C++ Objekte handelt (bei einer Kopie wäre das nicht so). Allerdings müssen wir explizit die Verbindung vom `if`-Block zum ersten Block in seinem `then`-Zweig herstellen, falls $T_{switch-head} = T_{cur}$ galt.
   Außerdem müssen wir vom Ende des Traces-Teils aus, der sich jetzt im `then`-Zweig befindet,



(a) `else-if` Struktur mit Trace (b) `else-if` am Trace-Ende neu gefaltet (c) `else-if` in der Trace-Mitte neu gefaltet

Abbildung 6.16.: Verschiedene Möglichkeiten der `else-if` Neufaltung

die Verbindung zum Trace-Block nach dem `if`, also dem `curBlock`, wieder herstellen. Diese beiden zu ergänzenden Verbindungen entsprechen exakt den beiden in Abbildung 6.14(b) sichtbaren Pfeilen.

7. **Deklaration von zugegriffenen Symbolen nach außen ziehen**
   Wie in Abbildung 6.14(b) am Beispiel des Symbol `j` zu sehen ist, müssen Symbole, auf die im kopierten Code zugegriffen wird, die aber erst im `switch` deklariert werden, in das Compound-Statement verschoben werden, das das `if` umgibt.

8. **Löschen des extrahierten `cases`**
   Falls es einen Fall-Through-Pfad zum `traceCase` gab, muß dieser Schritt übersprungen werden, ansonsten können wir den Teil vom `traceCase` bis zum nächstgelegenen `case` löschen, da er jetzt unausführbar ist. In Abbildung 6.14(b) wurden dabei z.B. die Statements `case 1:` und `j = work();` gelöscht.

9. **`IR_SelectionStmtConvertCounter` erhöhen**
   In die `IR_SelectionStmtConvertCounter`-Annotation des `switch` wird eingetragen, daß eine Konvertierung stattgefunden hat. Die Konvertierung des `switch` ist mit diesem Schritt abgeschlossen.

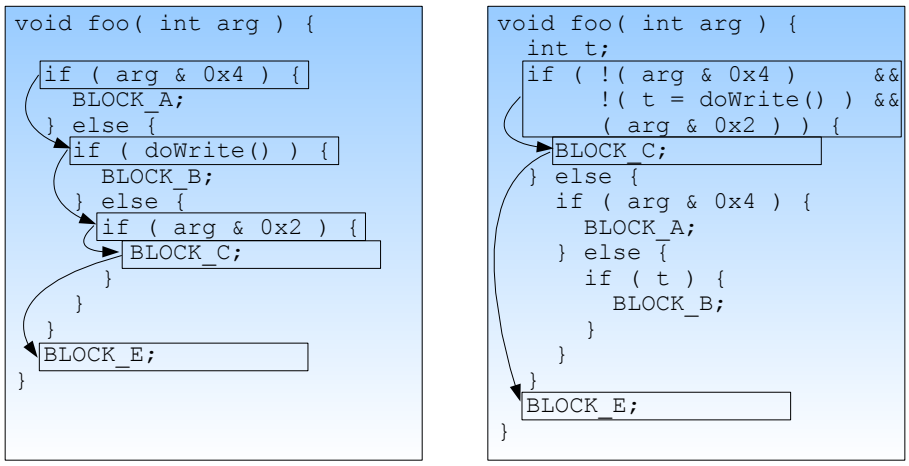
6.4.3. Neufaltung von `else-if` Strukturen

Die Neufaltung von `else-if` Strukturen ähnelt von ihrer Intention her der `switch`-Konvertierung. Auch hier soll Codegrößenwachstum begrenzt und der WCEP nach Möglichkeit verkürzt werden, allerdings nicht für den Fall, daß bei der Superblockbildung Code in ein `switch` verschoben werden soll, sondern für den Fall, daß er in eine `else-if` Struktur verschoben werden soll. Eine ***`else-if` Struktur*** ist dabei eine Folge $S = (ifelse_1, ..., ifelse_{n-1}, ifelse_n)$ von `IR_IfElseStmt`-Objekten, wobei für alle $i \in \{1, ..., n-1\}$ gilt, daß sich im `else`-Zweig von $ifelse_i$ ausschließlich das Statement $ifelse_{i+1}$ befindet (Beispiel: `if (..) {..} else if (..) {..} else {..}`). Zusätzlich ist es zulässig, daß $ifelse_n$ kein `IR_IfElseStmt`-Objekt, sondern ein `IR_IfStmt`-Objekt ist (Beispiel: `if (..) {..} else if (..) {..}`). Diese Fälle treten in realem Code relativ häufig auf.

Ein Beispiel für eine solche Struktur ist in Abbildung 6.16(a) skizziert. Wenn wir in dieser Situation die Superblockbildung durchführen würden, müssten wir `BLOCK_E` 3-mal kopieren. In Abbildung 6.16(b) ist dagegen die Situation nach der `else-if` Neufaltung am Ende des Traces dargestellt. Hier ist nur noch *eine* Kopie von `BLOCK_E` nötig um den Superblock zu bilden. Allerdings ist hier auch der bei der Neufaltung vorhandene Zielkonflikt sichtbar: Das Codegrößenwachstum wird eingeschränkt, allerdings geht dies in ungünstigen Fällen zu Lasten der WCET. Wenn z.B. in Abbildung 6.16(a) der Pfad durch die `else-if` Struktur, der beim `BLOCK_C` endet, eine fast ebenso hohe WCET besitzt, wie der Pfad, der bei `BLOCK_D` endet, dann wird durch die Transformation in Abbildung 6.16(b) zwar der Pfad zu `BLOCK_D` verkürzt, der Pfad zu `BLOCK_C` jedoch durch die zusätzlichen Bedingungsauswertungen im neu generierten `if` *verlängert*. Um ein Ansteigen der WCET zu vermeiden haben wir die Möglichkeit den Punkt der Neufaltung zu verschieben. Die Neufaltung ist dabei an jedem `else`-Zweig in $ifelse_2, ..., ifelse_{n-1}$ und am `then`- und `else`-Zweig von $ifelse_n$ möglich. Abbildung 6.16(c) zeigt zum Vergleich eine Neufaltung bei `if ( arg & 0x2 )` statt bei `BLOCK_D` wie in Abbildung 6.16(b). Die Neufaltung bei `if ( arg & 0x2 )` stellt hier einen guten Mittelweg dar: `BLOCK_E` muß hier immer noch 2-mal dupliziert werden, aber dafür ist hier eine Verlängerung des WCEP unwahrscheinlicher (abhängig von den konkreten WCET-Werten).

Genau wie bei der `switch`-Konvertierung muß auch hier schon während der Traceselektion bekannt sein an welchem Punkt wir eine bestehende `else-if` Struktur neu falten wollen, da dies das Codegrößenwachstum beeinflusst und die Traceselektion, wie in Abschnitt 6.3.2 erläutert, die Codegrößenzunahme vorraussagen und kontrollieren können muß. Um die Entscheidung über den Punkt der Neufaltung schon bei der Traceselektion treffen zu können, führt die Traceselektion bei jeder `else-if`-Struktur $S = (ifelse_1, ..., ifelse_{n-1}, ifelse_n)$ auf die sie bei der Simulation der Superblockbildung stößt, folgende Prozedur aus:

1. **Bestimme die Menge der möglichen Neufaltungspunkte**
   Dies sind diejenigen $ifelse_i \in S$ deren `else`-Zweig auf dem Trace liegt. Die Traceselektion weiß hierbei schon, wo der Trace verlaufen wird, da der Longest Path zum Zeitpunkt der Berechnung der Codegrößenzunahme schon feststeht, und der Trace nur entlang des Longest Path verlaufen darf.
2. **Bestimme, ob die Neufaltung zulässig ist**
   Nach jeder Neufaltung wird an allen $ifelse_i \in S$, deren Bedingung mit kopiert wurde, über die bereits bekannte `IR_SelectionStmtConvertCounter`-Annotation (s. Abbildung 6.15) festgehalten, wie oft dieses `if-else` bereits von einer Neufaltung betroffen war. Im aktuellen WCC gilt hier ein Grenzwert von 1, d.h. falls im Codeteil `if ( arg & 0x8 )` bis `BLOCK_C` in Abbildung 6.16(b) erneut ein Trace selektiert würde, so wäre die Neufaltung der verbleibenden `else-if` Struktur nicht mehr möglich. Die Neufaltung ist außerdem dann nicht zulässig, wenn innerhalb der neu zu faltenden `else-if`-Struktur bereits ein Superblock existiert, da wir sonst nicht sicherstellen könnten, daß dieser durch die Neufaltung nicht zerstört würde.
3. **Bestimme die initiale WCET**
   Die WCET des Gesamtprogramms wird über das IPET-Verfahren aus Kapitel 5 berechnet und als $wcet_{init}$ gespeichert.
4. **Bestimme den optimalen Neufaltungspunkt**
   Für *jeden* der möglichen Neufaltungspunkte $ifelse_i \in S$ wird:
   a) Die komplette `IR` kopiert. Hierbei ist einiger technischer Aufwand nötig, um die kopierten WCET-, Codegrößen- und Superblock-Annotationen in die kopierte `IR` zu übertragen, da alle dort enthaltenen Referenzen nach dem Kopieren noch auf die Objekte aus der originalen `IR` zeigen. Diese Referenzen müssen über eine Zuordnungstabelle so aktualisiert werden, daß sie auf die Objekte der kopierten `IR` zeigen.



(a) `else-if` Struktur mit Trace (b) `else-if` am Ende neu gefaltet

Abbildung 6.17.: `else-if` Neufaltung bei Bedingungen mit Schreibzugriffen

b) Die Neufaltung wird innerhalb der kopierten `IR` an dem gegebenen Punkt $ifelse_i$ ausgeführt.

c) Die resultierende WCET $wcet_i$ wird erneut per IPET-Verfahren berechnet.

5. **Auswertung**
   Unter den $ifelse_i \in S$, bei denen $wcet_i \leq wcet_{init}$ war, wird dasjenige $ifelse_i$ mit minimalem $wcet_i$ ausgewählt. In den `IR_SelectionStmtConvertCounter`-Annotationen aller `if-else`-Statements die von der Neufaltung mit betroffen sind wird eingetragen, daß sie für `eine` Neufaltung freigegeben sind.
   Falls kein solches $ifelse_i$ existiert, wird die Neufaltung nicht zugelassen, da wir sonst von einer Steigerung der WCET ausgehen müssten. Da die WCET unser primäres Optimierungskriterium ist (das sekundäre Kriterium ist die Codegröße) erzeugen wir hier auch keine Pareto-Front, sondern entscheiden uns direkt für den Punkt mit der minimalen berechneten WCET.

Nachdem die Traceselektion die `if-else`-Statements markiert hat, auf die die Neufaltung angewandt werden soll, muß die Superblockbildung diese Informationen nur noch an jeder `else-if` Struktur S aus den `IR_SelectionStmtConvertCounter`-Annotationen auslesen und die Neufaltung bis zum gewählten Punkt $ifelse_i \in S$ ausführen. Der Algorithmus zur Neufaltung, den auch die Traceselektion nutzt um den optimalen Neufaltungspunkt zu bestimmen, geht dabei folgendermaßen vor:

1. **Vorbereitung**
   Aus der einzigen Eingabe, dem `if-else`-Statement `refoldBaseIf`, das am weitesten außen liegt, bestimmen wir, ob es sich um eine `else-if` Struktur S handelt, und bis zu welchem Punkt $ifelse_{refold} \in S$ die Neufaltung von der Traceselektion eingeplant wurde.

2. **`if`-Bedingung generieren**
   Für das neu zu erzeugende `if` generieren wir eine Bedigung aus den Elementen $S_{refold} = (ifelse_1, ..., ifelse_{refold}) \subseteq S$. Wir erzeugen eine neue, leere Bedingung `traceCond` vom Typ `IR_Exp` und durchlaufen alle $ifelse_i \in S_{refold}$ in dieser Reihenfolge. An jedem $ifelse_i$ wird ein neuer Term der Bedingung generiert:

   a) Falls die Bedingung c_i `= cond(`$ifelse_i$`)` einen Schreibzugriff auf eine Variable[2] oder

[2] Hier benutzen wir die Def/Use-Sets aus Abschnitt 7.1.3 um Schreibzugriffe festzustellen.

einen Funktionsaufruf enthält, oder ihre Iteration-WCET[3] eine festgesetzte Grenze[4] überschreitet, so wird das Ergebnis der Bedingung zwischengespeichert um eine doppelte Auswertung zu verhindern. Hierzu legen wir eine neue Variable t_i mit dem Typ der Bedingung an und erzeugen einen neuen `IR_Exp`-Ausdruck $buffer_i$ mit dem Inhalt t_i `= copy(`c_i`)`. Die ursprüngliche Bedingung c_i wird durch eine Auswertung der Variablen t_i ersetzt und ihre Iteration-WCET wird über die bekannte Schätzungsheurstik aktualisiert. Eine solche Situation zeigt Abbildung 6.17(b). Ansonsten setzen wir $buffer_i = c_i$.

b) Falls der Trace-Nachfolger von $ifelse_i$ im `else`-Zweig von $ifelse_i$ liegt, müssen wir $term_i$ noch negieren, damit die neue Bedingung dem Trace "folgt". Dies ist z.B. in Abbildung 6.16(b) bei allen kopierten Bedingungen nötig gewesen. Wir erzeugen hierzu einen neuen Ausdruck $term_i$ `= !`$buffer_i$. Wir können auf diese Negation verzichten, falls der Trace-Nachfolger von $ifelse_i$ im `then`-Zweig liegt, dies ist allerdings nur beim letzten Element $ifelse_n$ möglich, wie z.B. in Abbildung 6.17 dargestellt. In diesem Fall ist $term_i = buffer_i$.

c) $term_i$ wird über den `&`-Operator an `traceCond` angehangen, wenn keine der Bedingungen Schreibzugriffe enthält oder Laufzeitfehler (Traps) verursachen kann, wie z.B. Integer-Divisionen, Gleitkomma-Operationen und Speicherzugriffe. Falls eine solche Bedingung vorhanden ist, muß der `&&`-Operator bei jedem $term_i$ benutzt werden, damit die Auswertung dieser Bedingungen nur dann erfolgt, wenn sie auch im Originalprogramm erfolgt wäre.

3. **Neues `if` erzeugen**
   Wir erzeugen ein neues `if-else`-Statement `ifNew` mit der generierten Bedingung `traceCond` und fügen `ifNew` vor dem `refoldBaseIf` in das umschließende Compound ein. `refoldBaseIf` wird danach in den `else`-Zweig von `ifNew` verschoben und die Anweisungen aus dem Trace-Nachfolger-Zweig von $ifelse_{refold}$ werden in den `then`-Zweig von `ifNew` verschoben.

4. **Aktualisieren der Annotationen**
   Für das neu eingefügte `ifNew` und das verschobene `refoldBaseIf` müssen abschließend noch die Annotationen aktualisert werden. Die WCET- und Codegrößen-Werte von `ifNew` werden über die bekannte Heuristik geschätzt und den Basisblockdaten hinzugefügt. Im Falle von `refoldBaseIf` ist eine Aktualisierung nur dann nötig, wenn es vorher nicht alleine in seinem Basisblock war, da es nur dann keine eigene WCET- und Codegrößen-Annotation trägt. Diese Annotationen werden in dem Fall erzeugt und die Daten wie üblich geschätzt.

   Falls das `refoldBaseIf` auf dem aktuell bearbeiteten Trace lag (Trace von `curBlock`, s. Algorithmus 6.1, Zeile 14/24/51), dann verläuft der Trace jetzt von `refoldBaseIf` zum ersten Block des `then`-Zweiges, ansonsten zum ersten Block des `else`-Zweiges. Diese Information legen wir in der `IR_SuperblockInfo`-Annotation des Blocks von `ifNew` ab.

5. **`IR_SelectionStmtConvertCounter` erhöhen**
   In die `IR_SelectionStmtConvertCounter`-Annotation aller $ifelse_i \in S_{refold}$ wird eingetragen, daß eine Konvertierung stattgefunden hat. Die Neufaltung der `else-if`-Struktur ist mit diesem Schritt abgeschlossen.

[3] Die Iteration-WCET der Bedingungen wird mit der Heuristik aus Abschnitt 5.2 abgeschätzt.
[4] Diese Grenze beträgt im aktuellen WCC 8 Zyklen, damit alle einfachen Vergleiche von der Zwischenspeicherung ausgeschlossen sind.

7. WCET-sensitive High-Level Superblockoptimierung

Die erzeugten Superblöcke können, wie bereits in [Chang et al., 1991] dargestellt, für weitere Optimierungen genutzt werden, ganz unanhängig von ihrem Nutzen für das Instruction Scheduling. Hierzu wurden als Teil der Diplomarbeit neue Analysemöglichkeiten innerhalb des WCC-Framework geschaffen und vorhandene erweitert (Abschnitt 7.1). Darauf aufbauend wurden die wichtigsten zwei der in [Chang et al., 1991] erwähnten Optimierungen implementiert, nämlich die Common Subexpression Elimination (Abschnitt 7.2) und die Dead Code Elimination bzw. Operation Migration (Abschnitt 7.3).

7.1. Analysewerkzeuge

Fast alle Compileroptimierungen benötigen genaue Informationen über die Auswirkungen einer Anweisung, z.B. darüber, welche Variablen von der Anweisung gelesen oder beschrieben werden. Insbesondere auf der High-Level-Ebene ist es nicht einfach zu bestimmen, ob z.B. eine bestimmte Anweisung einen Schreibzugriff auf ein Symbol vornimmt, da die High-Level-Statements beliebig komplexe Ausdrücke beinhalten können. Wir stellen daher im folgenden vier Analysetechniken und deren Umsetzung im WCC vor, die wir für die Formulierung der nachfolgenden Optimierungen benötigen werden.

7.1.1. Dominanzrelation

Die Dominanzrelation $dom : V \times V$ und die Postdominanzrelation $pdom : V \times V$ sind Relationen auf der Basisblockmenge V. $u\, dom\, v$ gilt dabei genau dann, wenn der Knoten u auf allen Pfaden von der Quelle des CFG nach v durchlaufen werden muß. Analog dazu gilt $v\, pdom\, u$ genau dann, wenn v auf jedem Pfad von u zu einer Senke des CFG durchlaufen werden muß. Beide Relationen sind ***partielle Ordnungen***, d.h. es kann u,v geben für die weder $u\, dom\, v$ noch $u\, pdom\, v$ gilt. Die Berechnung der Relationen ist bereits in die ICD-C IR integriert, so daß wir diese Informationen dort abrufen können.

7.1.2. Alias-Analyse

In fast allen Hochsprachen gibt es das Konzept der Referenzen und/oder der ***Zeiger*** (Pointer). Eine Zeigervariable stellt dabei kein eigenes Objekt im Speicher dar, sondern enthält nur die Adresse eines anderen Objekts, "zeigt" also darauf. Mit diesen Adressen kann der Programmierer dann wie mit normalen Ganzzahlen rechnen, was zu teilweise schwer lesbarem, aber sehr performantem, Code führen kann. ***Referenzen*** sind Alias-Namen, die es ermöglichen ein bestehendes Objekt unter einem neuen Namen anzusprechen. Sie zeigen damit für den gesamten Rest ihrer Existenz auf das anfangs festgelegte Zielobjekt, dieses muß jedoch nicht eindeutig bestimmt sein, es kann sich zum Beispiel um den Rückgabewert einer Funktion handeln. Beide Konzepte bergen aus Sicht des Compilers ein großes Problem: Es ist ohne weiteres nicht feststellbar auf welches Speicherobjekt eine Anweisung mit einer Zeiger- oder Referenzvariable zugreift, da dies erst zur Laufzeit endgültig feststeht. Man kann die Annahme treffen, daß ein Zeiger auf *alle* möglichen Speicherobjekte zeigen kann, allerdings ist dies eine sehr grobe Abschätzung, die die einzelnen Optimierungen in ihrer Entscheidungsfreiheit zu stark einschränkt. Daher benötigen wir eine Analyse, die uns sichere

Überabschätzungen für die Menge der Speicherobjekte liefert, auf die ein bestimmter Zeiger oder eine Referenz zeigen kann (Da der WCC ein C-Compiler ist, und die Sprache C nur das Konzept des Zeigers unterstützt, nicht aber das der Referenz, werden wir im folgenden nur noch von Zeigern sprechen.) Diese Analyse, die als Alias-Analyse bezeichnet wird, ist in verschiedenen Varianten bereits ausführlich erforscht worden. Die verschiedenen Varianten unterschieden sich dabei in den folgenden Kriterien:

- ***Intra-/Interprozeduralität***:
  Ob die Analyse nur innerhalb einer speziellen Funktion oder global auf der gesamten IR arbeitet. Im ersteren Fall müssen z.B. bei Funktionsaufrufen konservative Annahmen getroffen werden, die das Analyseergebnis verschlechtern können.
- ***Kontext-Sensitivität***:
  Ob Kontexte (s. Abschnitt 2.6) benutzt werden. Falls nicht, müssen alle Ergebnisse unabhängig von ihrem Kontext zusammengefasst werden. Hierbei können Informationen verloren gehen bzw. ungenauer werden.
- ***Fluß-Sensitivität***:
  Ob Alias-Beziehungen nur einmal global (bzw. pro Funktion bei einer intraprozeduralen Analyse) gespeichert werden, oder ob die Beziehungen an jeder einzelnen Anweisung ermittelt und gespeichert werden. Letzteres kann zu genaueren Ergebnissen führen.
- ***Feld-Sensitivität***:
  Ob ermittelt werden kann, daß ein Zeiger auf ein Feld einer Struktur (`struct`) / eines Arrays zeigt, oder ob nur gespeichert werden kann, daß er auf irgendeine Stelle der Struktur / des Arrays zeigt. Ebenso, ob festgehalten wird, daß Felder einer Struktur / eines Arrays auf eine Speicherstelle zeigen können, oder ob nur gespeichert wird, daß die Struktur / der Array insgesamt auf eine Speicherstelle zeigen kann.
- ***High-Level-/Low-Level-Analyse***:
  Auf welcher IR die Analyse durchgeführt wird. Auf der High-Level-IR ist dies wegen der dort noch vorhandenen Typinformationen normalerweise wesentlich einfacher.

Alias-Analyse im WCC

Wir benutzen in den folgenden Analysen eine im WCC bereits bestehende, interprozedurale, Kontext-, Fluß- und Feld-insensitive, High-Level-Alias-Analyse, die von Holger Bihr im Rahmen seiner Diplomarbeit [Bihr, 2005] in die ICD-C integriert wurde. Da die Aliasanalyse in der Fluß-sensitiven Variante im Allgemeinen unentscheidbar [Ramalingam, 1994], und in der Fluß-insensitiven Version immer noch NP-vollständig [Horwitz, 1997] und auch nicht polynomiell approximierbar [Chakaravarthy & Horwitz, 1986] ist, gibt es bei der im WCC integrierten Analyse die Möglichkeit, sie nach einer vorgegebenen Zeitschranke abzubrechen. Für diesen Fall und für den Fall das Scheiterns der Analyse müssen wir, wie oben schon angedeutet, pessimistische Annahmen über die Zeigerbeziehungen machen.

Objekte können im Speicher überlichweise auf dem ***Stack*** oder im ***Heap*** abgelegt werden. Im letzteren Fall spricht man von ***dynamisch allokiertem Speicher***. Da für die Allokation und Freigabe dieses Speichers normalerweise Betriebssystem-Funktionen benutzt werden (`malloc`, `calloc`, `free`, ...), die außerhalb der Reichweite unserer WCET-Analyse liegen, unterstützt der WCC momentan keinen dynamisch allokierten Speicher. Die einzigen Speicherobjekte, die damit noch das Ziel von Pointern sein können, sind daher lokale Variablen auf dem Stack oder globale Variablen auf dem Heap, die alle über ihren Namen, das ***Symbol***, eindeutig bezeichnet werden [ISO/IEC, 1999]. Da außerdem Zeiger selbst auch im Speicher abgelegt werden, genügt es, wenn wir in der Lage sind die Zeigerbeziehungen zwischen Symbolen zu analysieren. Im Falle einer zukünftigen Integration

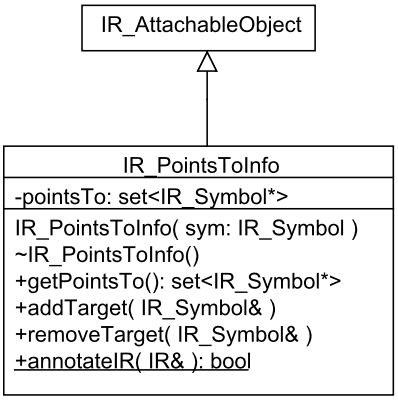


Abbildung 7.1.: Aufbau der `IR_PointsToInfo`-Klasse.

von dynamisch allokiertem Speicher in den WCC kann dieser Speicherbereich durch ein oder mehrere ausgezeichnete Symbole repräsentiert werden. Dadurch sind die Analysen auch auf diesen Fall erweiterbar.

Falls die Alias-Analyse Erfolg hat, können wir für jedes einzelne Symbol in der IR abfragen auf welche anderen Symbole es zeigen kann (***Points-To-Set***). Diese Darstellung der Zeigerbeziehungen wird auch als ***Points-To-Darstellung*** bezeichnet. Wie man sich leicht klarmacht bleibt die Gültigkeit der Points-To-Sets von unseren Optimierungen unberührt. Die Superblockbildung kopiert zwar Code, aber auf jedem Ausführungspfad werden danach dieselben Anweisungen ausgeführt wie vor der Superblockbildung, d.h. hier findet keine Beeinflussung der Points-To-Sets statt. Auch die noch vorzustellenden Optimierungen SB-CSE und SB-DCE führen keine neuen Zeigerbeziehungen ein, so daß die Überabschätzungen der Points-To-Sets weiterhin korrekt bleiben. Daher annotieren wir an jedes `IR_Symbol`, für das Alias-Informationen ermittelt wurden, die Ergebnisse der Alias-Analyse an dem Symbol. Dies wird mithilfe einer neuen Annotationsklasse `IR_PointsToInfo` (s. Abbildung 7.1) realisiert. Die Objekte dieser Klasse bleiben während der gesamten Optimierungskette erhalten, da wir in unseren Optimierungen niemals Symbole löschen sondern höchstens verschieben. Auch bei der Superblock Dead Code Elimination werden wir die Menge der realen Zeigerbeziehungen durch das Löschen von Statements höchstens verkleinern, die Gültigkeit der ermittelten Überabschätzung bliebt hiervon jedoch unberührt. Hierbei ist zu bemerken, daß wir höchstens Statements löschen, nicht jedoch die Symbole selbst.

7.1.3. Def/Use-Sets

Im Verlauf der Optimierungen ist es allerdings an vielen Stellen nicht ausreichend nur zu wissen, auf welche Symbole ein Zeiger deuten kann, sondern wir interessieren uns dabei vor allem für die Information darüber

- auf welche Symbole ein Ausdruck lesend zugreifen *kann* (***May Use-Set*** $\mathrm{USE}_{\mathrm{may}}$).
- auf welche Symbole ein Ausdruck lesend zugreifen *muß* (***Must Use-Set*** $\mathrm{USE}_{\mathrm{must}}$).
- auf welche Symbole ein Ausdruck schreibend zugreifen *kann* (***May Def-Set*** $\mathrm{DEF}_{\mathrm{may}}$).
- auf welche Symbole ein Ausdruck schreibend zugreifen *muß* (***Must Def-Set*** $\mathrm{DEF}_{\mathrm{must}}$).

Bei den ***May-Sets*** handelt es sich um Überabschätzungen, d.h. hier können mehr Symbole gelistet sein, als tatsächlich gelesen oder geschrieben werden, und bei den ***Must-Sets*** handelt es sich um Unterabschätzungen, bei denen jedes gelistete Symbol *garantiert* gelesen oder beschrieben wird.

```
...
int *p = &object;
int cond = work();
if ( *p && cond ) {

  int i;
  for ( i = 0; i < 10; i++ ) {
    doSth( i );
  }

}
...
```

Abbildung 7.2.: Beispielcode für Def-/Use-Sets.

`IR_Exp` E	USE_{may}(`IR_Exp` E, `bool` $useBranch$)
e_1 `=` e_2	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{false}) \cup \text{USE}_{\text{may}}(e_2, \texttt{true})$
e_1 ϕ`=` e_2	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{true}) \cup \text{USE}_{\text{may}}(e_2, \texttt{true})$
e_1 ψ e_2	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{true}) \cup \text{USE}_{\text{may}}(e_2, \texttt{true})$
$func$`(`e_1`, ..., `e_n`)`	$\leftarrow \bigcup_{i\in\{1,\ldots,n\}} \text{USE}_{\text{may}}(e_i, \texttt{true}) \cup \text{USE}_{\text{may}}(func)$
e_1`.`$field$	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{true}) \cup \{field\}$
e_1`->`$field$	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{true}) \cup \{field\} \cup \text{POINTS-TO}_{\text{may}}(e_1)$
e_1 `?` e_2 `:` e_3	$\leftarrow \bigcup_{i\in\{2,3\}} \text{USE}_{\text{may}}(e_i, useBranch) \cup \text{USE}_{\text{may}}(e_1, \texttt{true})$
c	$\leftarrow \emptyset$
e_1`[`e_2`]`	$\leftarrow \bigcup_{i\in\{1,2\}} \text{USE}_{\text{may}}(e_i, \texttt{true}) \cup \text{POINTS-TO}_{\text{may}}(e_1)$
`sizeof(`e_1`)`	$\leftarrow \emptyset$
`(type)(`e_1`)`	$\leftarrow \text{USE}_{\text{may}}(e_1, useBranch)$
π e_1	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{true})$
`*` e_1	$\leftarrow \text{USE}_{\text{may}}(e_1, \texttt{true}) \cup \text{POINTS-TO}_{\text{may}}(e_1)$
sym	$\leftarrow$ $useBranch = \texttt{true}$: $\{sym\}$; $useBranch = \texttt{false}$: $\emptyset$

Tabelle 7.1.: Regeln zur Berechnung von May Use-Sets USE_{may} von Ausdrücken

Alle Sets wollen wir nicht nur für Ausdrücke, sondern auch für Anweisungen und ganze Funktionen berechnen können. Beispiele für Def-/Use-Sets in Abbildung 7.2 sind z.B.

$$\text{DEF}_{\text{may}}(\texttt{cond = work();}) = \{\texttt{cond}\} \cup \text{DEF}_{\text{may}}(\texttt{work})$$
$$\text{USE}_{\text{must}}(\texttt{if (*p \&\& cond)}) = \{\texttt{p}, \texttt{object}\}$$

Die ICD-C IR bietet zwar bereits die Möglichkeit, über Ausdrücke und ihre Unterausdrücke zu iterieren, und die bei der Iteration durchlaufenen Symbol-Ausdrücke als Def oder Use zu klassifizieren, allerdings gibt es noch keine vorgefertigte Möglichkeit, diese Ergebnisse in Form eines Def/Use-Sets zusammenzufassen, und auch keine Möglichkeit, die Ergebnisse der Alias-Analyse bei der Untersuchung mit einfließen zu lassen. Zur Berechnung der Def/Use-Sets verfolgen wir hier eine Strategie, die auch von Tonella in [Tonella, 1999] angewandt wird. Bei der Berechnung eines Def- oder Use-Sets für Ausdrücke richten wir uns nach dem syntaktischen Aufbau der Ausdrücke (***Syntax Directed Definitions***) und wenden entsprechende Regeln an, um das Def- oder Use-Set zu erhalten.

Um die Nutzung der Def-/Use-Sets zu vereinfachen, wurde eine Klasse `IR_DefUseSetContainer`

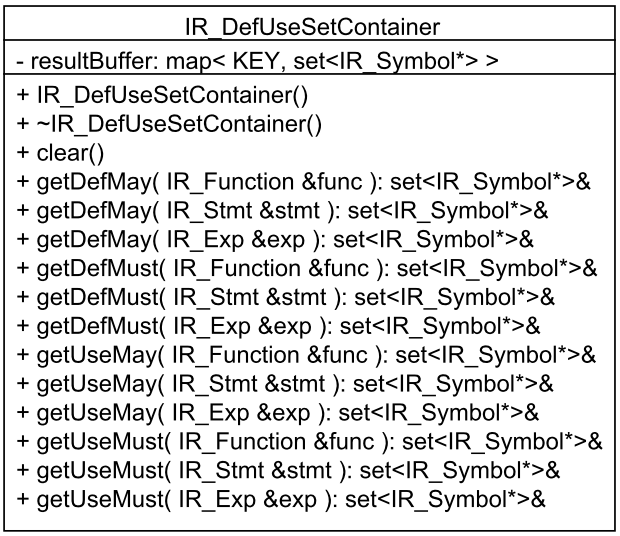


Abbildung 7.3.: Aufbau der `IR_DefUseSetContainer`-Klasse.

geschaffen, die diese Regeln umsetzt und die abgefragten Def-/Use-Sets speichert, da diese dynamisch allokiert werden und somit eine potentielle Quelle von Speicherlecks darstellen. Falls die zugrundeliegenden Ausdrücke und Anweisungen verändert werden, muß dies dem entsprechenden Objekt der Klasse `IR_DefUseSetContainer` über die Methode `clear()` mitgeteilt werden (siehe Abbildung 7.3). Dadurch werden alle gespeicherten Def-/Use-Sets gelöscht. Durch die Speicherung ist sichergestellt, daß mehrfache Anfragen nach derselben Information, wie sie während der Analysen oft vorkommen werden, nicht zu wiederholten, redundanten Berechnungen führen. Außerdem wird die Speicherverwaltung vereinfacht, da der Container in seinem Destruktor das Löschen der erzeugten Sets übernimmt.

Die Regeln für die Bestimmung von May Use-Sets sind in Tabelle 7.1 dargestellt. Hierbei sind $\phi \in \{$`+`, `-`, `*`, `/`, `%`, `«`, `»`, `&`, `|`, `^`$\}$, $\psi \in \{$`+`, `-`, `*`, `/`, `%`, `«`, `»`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `|`, `^`, `&&`, `||`, `,`$\}$ und $\pi \in \{$`+`, `-`, `!`, `~`, `&`, `++`, `--`$\}$ als Platzhalter für die jeweiligen Operatoren im Text enthalten. c und sym stehen für eine beliebige Konstante bzw. ein beliebiges Symbol. Die USE_{may}-Funktion hat hier noch ein zweites Argument $useBranch$, das angibt, ob Symbole, die im aktuellen Aufruf von USE_{may} gefunden werden, als "gelesen" betrachtet werden müssen. Dies ist nötig, da sich auch in Ausdruckszweigen, die definiert werden, Uses verbergen können. Bei der Analyse von `(c ? i : j) = 100` würden wir z.B. zuerst mit $useBranch =$ `false` in den Ausdruck `c ? i : j` absteigen (1. Zeile von Tabelle 7.1). Dort würden wir dann auf den `?:`-Operator stoßen, in dessen Bedingung (im Beispiel wäre das der Ausdruck `c`) wir wieder mit $useBranch =$ `true` absteigen (7. Zeile von Tabelle 7.1). Auf diese Weise finden wir die Benutzung von Symbol `c`, wohingegen die Symbole `i` und `j` nicht als Use zählen, da bei ihnen während der Abarbeitung des `?:`-Operators der Wert von $useBranch$ (`false`) beibehalten wird und sie deshalb nicht als Use gewertet werden (letzte Zeile von Tabelle 7.1). Alle folgenden Funktionen zur Berechnung der Def-/Use-Sets verfolgen Funktionsaufrufe $func$ über Aufrufe von z.B. $\text{DEF}_{\text{may}}(func)$ weiter. Dies würde in dieser naiven Form dazu führen, daß bei der Analyse eines rekursiven Programms z.B. unendlich oft DEF_{may} aufgerufen würde. Um dieses Problem zu vermeiden, verwalten alle Analysefunktionen einen Call-Stack, der aktualisiert wird, wenn die Analyse in eine neue Funktion absteigt. Neue Funktionen, die bereits auf dem Call-Stack liegen werden dabei nicht erneut betrachtet, wodurch die potentielle Endlosschleife aufgelöst wird. Diese Call-Stacks werden als Parameter der Funktionen weitergereicht, da sie die Notation jedoch schlechter lesbar machen, werden sie in den Tabellen nicht aufgeführt. Die

`IR_Exp` E	DEF_{may}(`IR_Exp` E)
e_1 `=` e_2	$\leftarrow \text{L-VALUE}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_2)$
e_1 ϕ`=` e_2	$\leftarrow \text{L-VALUE}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_2)$
e_1 ψ e_2	$\leftarrow \text{DEF}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_2)$
$func$`(`e_1`, ..., `e_n`)`	$\leftarrow \bigcup_{i\in\{1,\ldots,n\}} \text{DEF}_{\text{may}}(e_i) \cup \text{DEF}_{\text{may}}(func)$
e_1`.`$field$	$\leftarrow \text{DEF}_{\text{may}}(e_1)$
e_1`->`$field$	$\leftarrow \text{DEF}_{\text{may}}(e_1)$
e_1 `?` e_2 `:` e_3	$\leftarrow \bigcup_{i\in\{1,\ldots,3\}} \text{DEF}_{\text{may}}(e_i)$
c	$\leftarrow \emptyset$
e_1`[`e_2`]`	$\leftarrow \text{DEF}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_2)$
`sizeof(`e_1`)`	$\leftarrow \emptyset$
$\pi(\notin \{$`++, --`$\})$ e_1	$\leftarrow \text{DEF}_{\text{may}}(e_1)$
`*` e_1	$\leftarrow \text{DEF}_{\text{may}}(e_1)$
`++` e_1	$\leftarrow \text{L-VALUE}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_1)$
e_1 `++`	$\leftarrow \text{L-VALUE}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_1)$
`--` e_1	$\leftarrow \text{L-VALUE}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_1)$
e_1 `--`	$\leftarrow \text{L-VALUE}_{\text{may}}(e_1) \cup \text{DEF}_{\text{may}}(e_1)$
sym	$\leftarrow \emptyset$

Tabelle 7.2.: Regeln zur Berechnung von May Def-Sets DEF_{may} von Ausdrücken

Methoden, die im folgenden eingeführt werden, sind wie z.B. $\text{USE}_{\text{may}}(E, useBranch)$ private Methoden von `IR_DefUseSetContainer`. Für Aufrufe von außen werden Kapselungs-Methoden (z.B. `getUseMay( IR_Exp &exp )`, s. Abbildung 7.3) angeboten, die die fehlenden Parameter korrekt initialisieren. *useBranch* wird hier z.B. so initialisiert, daß angegeben wird, ob der Ausdruck, für den die Methode aufgerufen wurde, seinen Inhalt ausliest. Diese Kapselungs-Methoden erzeugen auch einen leeren Call-Stack für die oben erwähnte Behandlung von rekursiven Programmen.

Bei der Bestimmung der May Def-Sets gehen wir etwas anders vor. Jeder C-Ausdruck, der eine Definition enthält, setzt die Existenz eines so genannten ***L-Value***-Ausdrucks vorraus, der das Ziel der Definition ist. Dieser Ausdruck muß ein nicht-konstantes Speicherobjekt identifizieren, dessen Inhalt neu geschrieben werden soll. In unserem Fall sind dies die von einem Ausdruck beschriebenen Symbole, im folgenden auch als ***L-Values*** bezeichnet. Wir benutzen zur Berechnung der DEF_{may}-Funktion wieder den regelbasierten Ansatz aus [Tonella, 1999], allerdings benutzen wir hier keine Hilfsvariable, die uns angibt, ob wir uns in einem Ausdruckszweig befinden, dessen L-Value definiert wird, sondern wir ermitteln bei jedem Operator, der eine Definition darstellt, welcher L-Value an der jeweiligen Stelle definiert wird. Hierzu benutzen wir eine Funktion $\text{L-VALUE}_{\text{may}}$, die berechnet, welche L-Values von einem gegebenen Ausdruck identifiziert werden. Der Regelsatz für DEF_{may} ist in Tabelle 7.2 und derjenige für $\text{L-VALUE}_{\text{may}}$ in Tabelle 7.3 abgebildet.

Für die Auflösung von Pointern benötigen wir bei fast allen Regelsätzen in den Regeln für die `->`, `[]` und `*` Operatoren die Ergebnisse der Alias-Analyse aus Abschnitt 7.1.2. Für eine einfache Formulierung benötigen wir allerdings eine Auswertungsfunktion $\text{POINTS-TO}_{\text{may}}$, die die Menge von Symbolen bestimmt, auf die ein *Ausdruck* zeigen kann. Die Alias-Analyse selbst liefert uns nur die Information worauf ein bestimmtes *Symbol* zeigen kann. Daher benutzen wir auch für die Berechnung der $\text{POINTS-TO}_{\text{may}}$-Funktion einen Regelsatz wie er in Tabelle 7.4 aufgelistet ist. λ ist hierbei ein Operator aus {`+`, `-`}, c ist eine beliebige Konstante und v eine beliebige ganzzahlige Variable. Im Falle von Zuweisungen oder einfachen Additionen/Subtraktionen auf einem Pointer geben wir das Points-To-Set des Zuweisungsziels oder des Pointer-Operanden zurück. Bei Funk-

IR_Exp E	L-VALUE$_{\text{may}}$(IR_Exp E)
e_1 = e_2	$\leftarrow$ L-VALUE$_{\text{may}}(e_1)$
e_1 ϕ= e_2	$\leftarrow$ L-VALUE$_{\text{may}}(e_1)$
e_1 $\psi(\neq ,)$ e_2	$\leftarrow \emptyset$
e_1, e_2	$\leftarrow$ L-VALUE$_{\text{may}}(e_2)$
$func$(e_1, ..., e_n)	$\leftarrow \emptyset$
e_1.$field$	$\leftarrow$ L-VALUE$_{\text{may}}(e_1)$
e_1->$field$	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
e_1 ? e_2 : e_3	$\leftarrow \bigcup_{i\in\{2,3\}}$ L-VALUE$_{\text{may}}(e_i)$
c	$\leftarrow \emptyset$
e_1[e_2]	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
(type)(e_1)	$\leftarrow$ L-VALUE$_{\text{may}}(e_1)$
$\pi(\notin \{*, \text{(type)}\})$ e_1	$\leftarrow \emptyset$
* e_1	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
sym	$\leftarrow \{sym\}$

Tabelle 7.3.: Regeln zur Berechnung von May L-Value-Sets L-VALUE$_{\text{may}}$ von Ausdrücken

tionsaufrufen überprüfen wir alle `return`-Anweisungen in der aufgerufenen Funktion und geben die Vereinigung von deren Points-To-Sets zurück. Im Falle der `.` und `->` Operatoren müssen wir für die Bestimmung der Points-To-Ziele nur das Points-To-Set des Basisausdrucks e_1 ermitteln, da unsere Alias-Analyse nicht Feld-sensitiv ist. Alle Pointer-Ziele aller Felder der Struktur werden daher in der Alias-Analyse als Pointer-Ziele der Struktur selbst gelistet. Für den `->`, `[]` und `*` Operator müssen wir außerdem eine doppelte Auflösung vornehmen: Zuerst werden die Symbole ermittelt auf die der Basisausdruck zeigen kann, und dann die Points-To-Ziele dieser Symbole, da wir das Points-To-Set des Gesamtausdrucks berechnen müssen. Falls wir auf konstante Ausdrücke c treffen, müssen wir davon ausgehen, daß dieser Ausdruck auf *alle* im Programm verfügbaren Symbole zeigen kann (Symbolmenge S), da wir beim Zugriff auf eine konstante Adresse keine Möglichkeit haben dies in der Alias-Analyse zu erfassen. Beim Adress-Operator `&` können wir mithilfe der bereits vorgestellten L-VALUE$_{\text{may}}$-Funktion einfach die Menge der enthaltenen L-Values zurückgeben. Wenn die Ermittlung der Points-To Sets schließlich auf ein Symbol trifft, so kann sie entweder die Ergebnisse der Alias-Analyse zurückgeben, falls diese vorhanden sind, oder muß eine konservative Überabschätzung cons_approx(sym) zurückgeben, in die nach dem ***Address-Taken-Schema*** alle Symbole aufgenommen werden, die:

- ein Array-Symbol sind
- deren Adresse an beliebiger Stelle mit dem `&`-Operator enthüllt wurde
- ein globales Symbol sind

Der letzte Punkt ist nur nötig, falls die IR nicht als ***vollständig*** markiert wurde, falls also noch weitere, unbekannte Codeteile existieren, die erst später hinzugelinkt werden. In diesem Fall kann es in diesen unbekannten Codeteilen weitere `&`-Operatoren geben.

Die Berechnung der Must Def/Use-Sets funktioniert fast exakt genauso wie die der May Def/Use-Sets. Wir müssen dabei solche Defs und Uses, die von bedingten Ausdrucksteilen (e_2 in e_1 `&&` e_2; e_2 in e_1 `||` e_2; e_2 und e_3 in e_1 `?` e_2 `:` e_3) verursacht werden aus den Must Def/Use-Sets ausschließen. Außerdem ist es nötig, diejenigen L-Values, die nicht eindeutig bestimmt werden konnten von den Must Def-Sets auszuschließen. Dasselbe trifft auch auf Points-To-Ziele zu, die während der Berechnung der Must Use-Sets nicht eindeutig bestimmt werden konnten. Die Eindeutigkeit ist dabei immer dann gegeben, wenn nur ein Element in der DEF$_{\text{may}}$- oder USE$_{\text{may}}$-Menge vorhanden

IR_Exp E	POINTS-TO$_{\text{may}}$(IR_Exp E)
e_1 = e_2	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
e_1 ϕ= e_2	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
e_1, e_2	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
e_1 λ c	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
c λ e_1	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
e_1 λ v	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
v λ e_1	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
$func(e_1, \ldots, e_n)$	$\leftarrow \bigcup_{(\texttt{return } e_i) \in func}$ POINTS-TO$_{\text{may}}(e_i)$
$e_1.field$	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
e_1->$field$	$\leftarrow \bigcup_{s_i \in \text{POINTS-TO}_{\text{may}}(e_1)}$ POINTS-TO$_{\text{may}}(s_i)$
e_1 ? e_2 : e_3	$\leftarrow \bigcup_{i \in \{2,3\}}$ POINTS-TO$_{\text{may}}(e_i)$
c	$\leftarrow S$
e_1[e_2]	$\leftarrow \bigcup_{s_i \in \text{POINTS-TO}_{\text{may}}(e_1)}$ POINTS-TO$_{\text{may}}(s_i)$
sizeof(e_1)	$\leftarrow \emptyset$
$\pi(\notin \{\&, *\})e_1$	$\leftarrow$ POINTS-TO$_{\text{may}}(e_1)$
& e_1	$\leftarrow$ L-VALUE$_{\text{may}}(e_1)$
* e_1	$\leftarrow \bigcup_{s_i \in \text{POINTS-TO}_{\text{may}}(e_1)}$ POINTS-TO$_{\text{may}}(s_i)$
sym	$\leftarrow$ Alias-Analyse-Ergebnis A vorhanden: A; sonst: cons_approx(sym)

Tabelle 7.4.: Regeln zur Berechnung von May Points-To-Sets POINTS-TO$_{\text{may}}$ von Ausdrücken.

ist. Abgesehen von diesen Abweichungen werden für die Berechnung der Must Def/Use-Sets die bereits vorgestellten Algorithmen verwendet.

Aufbauend hierauf können wir die Sets pro Anweisung berechnen, indem wir die Sets aller in der Anweisung direkt enthaltenen Ausdrücke aufsummieren. Ein Ausdruck ist dabei ***direkt*** in einer Anweisung enthalten, wenn er ein direktes Unterobjekt der Anweisung ist. Im Beispielcode `if ( *p && cond ) { BLOCK }` ist z.B. `*p && cond` ein direktes Unterobjekt des `if`. `*p`, `p`, `cond` und alle Ausdrücke innerhalb von `BLOCK` sind dies jedoch *nicht*. Die Summation ist bei den May-Sets offensichtlich korrekt, da wir hier eine Überabschätzung berechnen. Bei den Must-Sets müssen wir beachten, daß wir dort nur die Must-Sets solcher Ausdrücke aufsummieren dürfen, die immer dann ausgewertet werden, wenn das Statement ausgeführt wird. In C gibt es allerdings nur einen solchen Fall in dem dies nicht zutrifft, nämlich beim "`i++`-Teil" von `for`-Statements. Die Must-Sets dieses Ausdrucks werden daher nicht Teil der Must-Sets des gesamten `for`-Statements.

Die Sets für die gesamten Funktionen können wir ebenfalls analog über die Summierung der entsprechenden Sets aller Statements innerhalb der Funktion erhalten, wobei wir bei den Must-Sets hier nur solche Statements betrachten dürfen, die in der Funktion unbedingt ausgeführt werden, deren Basisblock also den Funktionskopf postdominiert. Must-Uses und Must-Defs von Statements, die z.B. innerhalb einer `if`-Verzweigung ausgeführt werden, werden auf diese Weise von den Must-Uses oder Must-Defs der Funktion ausgeschlossen.

7.1.4. Lebensdauer-Analyse

Die letzte verbleibende Analyse die wir für die Superblockoptimierungen benötigen werden und die in dieser Form noch nicht in der ICD-C IR vorhanden ist, ist eine Lebensdauer- oder Livetime-Analyse. Eine Symbol wird dabei als *vor* einer Anweisung s ***lebendig*** (***live-in***) bezeichnet, falls

es einen Pfad P von s zur einer Senke des CFG gibt (s eingeschlossen), auf dem der Wert des Symbols ausgelesen bzw. benutzt wird. Andernfalls ist das Symbol *vor* dieser Anweisung ***tot***. Analog lässt sich auch die Lebendigkeit *nach* einer Anweisung s definieren (***live-out***), hierbei sind dieselben Pfade P exklusive s zugrunde zu legen. Das Ziel der Lebensdauer-Analyse ist, festzustellen welche Symbole beim Betreten bzw. Verlassen einer Anweisung jeweils lebendig sind. Die ICD-C IR bietet bereits die Berechnung von Def/Use-Ketten an, aus denen die Lebensdauer-Informationen extrahiert werden können, allerdings berücksichtigt diese Darstellung ebenfalls nicht die Ergebnisse der Alias-Analyse und ist nicht in der Lage Funktionsaufrufe zu verfolgen. Da beides mit den im letzten Abschnitt vorgestellten Def/Use-Sets möglich ist und die Umsetzung der Lebensdauer-Analyse mit diesen Sets relativ einfach ist, werden wir im folgenden statt der Def/Use-Ketten die in diesem Abschnitt vorgestellte Lebensdauer-Analyse benutzen.

Datenflußanalyse

Da die Lebensdauer-Analyse auf dem allgemeinen Konzept der Datenflußanalyse aufbaut, soll dieses hier kurz vorgestellt werden. Weiterführende Erläuterungen sind in [Cousot & Cousot, 1977], [Muchnick, 1997] und [Rüthing, 2008] zu finden. Alle Datenflußanalysen haben gemeinsam, daß die zu ermittelnden Informationen Elemente eines vollständigen Verbandes sein müssen.

Definition 1. *Eine* **partielle Ordnung** $(L, \sqsubseteq)$ *ist ein Tupel aus*

- *einer Menge* L
- *einer binären Relation* $\sqsubseteq \subseteq L \times L$ *in der die folgenden Eigenschaften gelten:*
  1. *Reflexivität:* $\forall X \subseteq L : (X, X) \in \sqsubseteq$
  2. *Antisymmetrie:* $\forall X, Y \subseteq L : (X, Y) \in \sqsubseteq \wedge (Y, X) \in \sqsubseteq \Rightarrow X = Y$
  3. *Transitivität:* $\forall X, Y, Z \subseteq L : (X, Y) \in \sqsubseteq \wedge (Y, Z) \in \sqsubseteq \Rightarrow (X, Z) \in \sqsubseteq$

In der partiellen Ordnung ist $\sqcup X$ *das* **Supremum** *einer Menge* $X \subseteq L$*, falls* $\forall x \in X : x \sqsubseteq \sqcup X$ *gilt, und* $\sqcap X$ *ist das* **Infimum** *von* $X \subseteq L$*, falls* $\forall x \in X : \sqcap X \sqsubseteq x$ *gilt.*

Definition 2. *Ein* **vollständiger Verband** *ist eine partielle Ordnung* $(L, \sqsubseteq)$ *in der das Supremum und das Infimum zu jedem* $X \subseteq L$ *existieren. Damit existiert in einem vollständigen Verband auch automatisch ein kleinstes Element* $\bot = \sqcap L$ *und ein größtes Element* $\top = \sqcup L$*.*

Ein triviales Beispiel für einen solchen vollständigen Verband ist $(Pot(S), \subseteq)$, bestehend aus der Potenzmenge der Symbole eines Programms und dem üblichen Mengen-Vergleichsoperator $\subseteq$. Die Eigenschaften einer partiellen Ordnung sind hier offensichtlich gegeben und das Supremum bzw. Infimum von $X \subseteq Pot(S)$ entspricht der Vereinigung bzw. dem Schnitt der Elemente aus X. $\bot$ ist hier die leere Menge $\emptyset$ und $\top$ ist gleich S. Genau dies ist auch der Verband der uns im folgenden interessieren wird, denn alle unsere Ergebnisse (die Mengen lebendiger Variablen) sind Elemente des Verbandes. Die vollständigen Verbände finden allerdings erst im Rahmen eines Datenfluß-Analyse-Framework eine Anwendung.

Definition 3. *Ein* **monotones Datenfluß-Analyse-Framework** $((L, \sqsubseteq), F)$ *besteht aus*

- *einem vollständigen Verband* $(L, \sqsubseteq)$
- *einer Menge* F *von monotonen Funktionen* $f : L \to L$ *für die gilt, daß*
  1. *die Identitätsfunktion id mit* $\forall X \subseteq L : id(X) = X$ *in* F *enthalten ist*
  2. *die Menge* F *abgeschlossen ist. Die Verkettung* $f \circ g$ *muß für alle Funktionen* $f, g \in F$ *in* F *enthalten sein.*

Als **Instanz** I *eines monotonen Datenfluß-Analyse-Framework* $((L, \sqsubseteq), F)$ *bezeichnen wir ein Tupel* $(V, E, v_0, init)$ *mit*

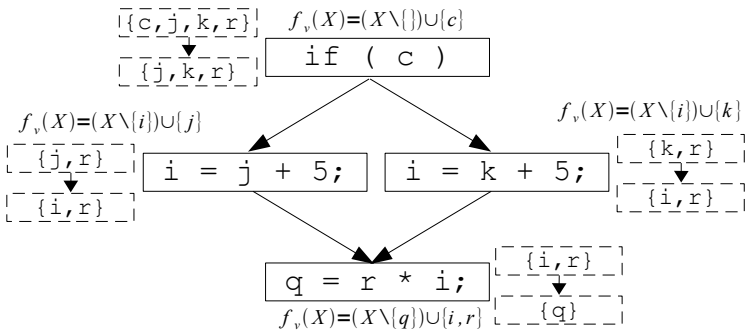


Abbildung 7.4.: Beispiel für die Arbeitsweise der Transferfunktionen.

- *einem Kontrollflußgraphen G=(V,E), dessen Knoten $v \in V$ mit jeweils einer Transferfunktion $f_v \in F$ annotiert sind,*
- *einem Startknoten v_0 und*
- *der Information init, die an v_0 gilt.*

Bei der Lebensdauer-Analyse verwenden wir das Framework $((Pot(S), \subseteq), F_{live})$, wobei jede Funktion $f_v \in F_{live}$ für jeden Knoten $v \in Pot(S)$ von der Form $f_v(X) = (X \setminus \text{kill}_v) \cup \text{gen}_v$ ist. kill_v ist dabei die Menge von Symbolen, die am Knoten v beschrieben werden, und gen_v ist die Menge der Symbole, deren Inhalt dort gelesen wird. DFA-Frameworks mit so strukturierten Transferfunktionen sind besonders gut zu lösen und werden als ***Bitvektorprobleme*** bezeichnet, da die Transferfunktion als einfache Funktion auf einem Bitvektor realisiert werden kann. In unserem Fall stände dabei jedes Bit für die Lebendigkeit einer bestimmten Variable. Das Ziel der Analyse ist, die Lebendigkeitsinformationen $\text{LIVE-IN}_{\text{may}}$ und $\text{LIVE-OUT}_{\text{may}}$ an *jedem* Knoten des CFG zu berechnen. Dazu wird, ausgehend vom Startknoten, aus der dort gegebenen $\text{LIVE-OUT}_{\text{may}}$-Information *init* über die Transferfunktion die $\text{LIVE-IN}_{\text{may}}$-Information des Knoten berechnet. Diese $\text{LIVE-IN}_{\text{may}}$-Information wird über die Supremumsbildung (hier: Mengenvereinigung) in die $\text{LIVE-OUT}_{\text{may}}$-Information aller Vorgänger des Knotens übernommen. Ein anschauliches Beispiel für die Anwendung der Transferfunktionen findet sich in Abbildung 7.4. Die Startinformation für die Analyse dieses Programmabschnitts sei die $\text{LIVE-OUT}_{\text{may}}$-Information {`q`} am Knoten `q = r * i`, `r` und `i` seien also nach diesem Knoten tot. Ausgehend hiervon wird jeweils die Transferfunktion angewandt und die so erhaltene $\text{LIVE-IN}_{\text{may}}$-Information an die Vorgängerknoten propagiert. Die Lebensdauer-Analyse gehört daher zur Klasse der ***Rückwärts-Analysen***, im Gegensatz zu den ***Vorwärts-Analysen*** die ihre Informationen in Richtung des Kontrollflusses propagieren.

Wir werden die Lebensdauer-Analyse immer nur lokal in einer Funktion durchführen. Die Instanzen I, auf denen wir arbeiten, haben als Knotenmenge V die Menge der Statements und die Kontrollflußkanten zwischen den Statements bilden die Kantenmenge E. Da bei der Analyse die Kanten im Funktions-CFG nur *rückwärts* durchlaufen werden, ist der Startknoten unserer Instanzen immer ein virtueller Endknoten der Funktion, der eine eingehende Kante von jeder Senke des Funktions-CFGs besitzt. Die Information an diesem Startknoten ist eine Menge von Symbolen, von denen wir annehmen müssen, daß sie außerhalb der Funktion noch lebendig sind. Die Menge besteht aus:

- globalen Symbolen,
- statischen Symbolen aus der Funktion und
- den Points-To-Sets von Funktionsargumenten, die Zeiger sind.

```
1  // Initialisiere die Worklist und die Knoteninformationen
2  worklist = V \ {u_0};
3  for ( v ∈ V ) {
4    if ( v == v_0 ) {
5      node_in[v] = init;
6      node_out[v] = init;
7    } else {
8      node_in[v] = ⊥;
9      node_out[v] = ⊥;
10   }
11 }
12
13 // Worklist abarbeiten
14 while ( worklist ≠ ∅ ) {
15   v = worklist.pop();
16
17   for ( (v,u) ∈ E ) {
18     node_out[v] = node_out[v] ⊔ node_in[u];
19   }
20   node_in[v] = f_v(node_out[v]);
21
22   if ( node_in[v] or node_out[v] changed ) {
23     worklist.append( {u|(u,v) ∈ E} );
24   }
25 }
```

Algorithmus 7.1: Worklist-Algorithmus zur Fixpunktberechnung bei Backward-Analyse.

```
1  // Initialisiere die Worklist und die Knoteninformationen
2  worklist = topsort(V \ {u_0});
3  for ( v ∈ V ) {
4    if ( v == v_0 ) {
5      LIVE-IN_may[v] = global_syms ∪ static_syms ∪ pointer_argument_targets;
6      LIVE-OUT_may[v] = global_syms ∪ static_syms ∪ pointer_argument_targets;
7    } else {
8      LIVE-IN_may[v] = ∅;
9      LIVE-OUT_may[v] = ∅;
10   }
11 }
12
13 // Worklist abarbeiten
14 while ( worklist ≠ ∅ ) {
15   v = worklist.pop();
16
17   for ( (v,u) ∈ E ) {
18     LIVE-OUT_may[v] = LIVE-OUT_may[v] ∪ LIVE-IN_may[u];
19   }
20   LIVE-IN_may[v] = ( LIVE-OUT_may[v] \ filterAS(DEF_must(v)) ) ∪ USE_may(v);
21
22   if ( LIVE-IN_may[v] or LIVE-OUT_may[v] changed ) {
23     worklist.append( {u|(u,v) ∈ E} );
24   }
25 }
```

Algorithmus 7.2: Konkreter Worklist-Algorithmus zur Durchführung der Lebensdauer-Analyse.

Die Anwendung der Transferfunktionen und die Propagation der errechneten Lebendigkeitsinformationen zu den Vorgängerknoten wird solange wiederholt, bis sich ein ***Fixpunkt*** einstellt, bis also keine weiteren Änderungen der Informationen an den Knoten mehr eintreten. Die Notwendigkeit der Fixpunktberechnung ergibt sich hierbei daraus, daß der CFG Schleifen enthalten kann, entlang derer die Informationen zyklisch aktualisiert werden können. Der hierbei verwendete ***Worklist***-Algorithmus ist in Listing 7.1 in der allgemeinen Form einer iterativen Datenfluß-Analyse beschrieben. In Listing 7.2 findet sich eine angepasste Version des Algorithmus, die das Lebensdauer-Framework löst. `filterAS` ist hier eine Funktion, die Array- und Struct-Symbole aus einer gegebenen Symbolliste löscht und die verbleibende Symbolliste zurückgibt. Dies ist in unserem speziellen Fall nötig, da wir ein Symbol nur dann aus der Liste der lebendigen Symbol löschen dürfen, wenn wir sicherstellen können, daß sein Inhalt im aktuellen Statement v überschrieben wird. Da unsere Alias-Analyse und auch die Def/Use-Sets nicht Feld-sensitiv sind, können wir bei beschriebenen Array- oder Struct-Symbolen nicht sicherstellen, daß *alle* Teile des Arrays oder der Struktur überschrieben wurden bzw. welche davon noch lebendig sein können.

Das DFA-Framework ist monoton, da die Transferfunktionen monoton sind: Im Verlauf des Algorithmus wird aus keinem der Live-In- oder Live-Out-Sets ein einmal eingefügtes Element je wieder entfernt. Aus dieser wichtigen Eigenschaft der Monotonie folgt zusammen mit der Tatsache, daß die Größe der Live-In- bzw. Live-Out-Sets durch $|S|$ beschränkt ist, direkt die Terminierung des Algorithmus. Da der Algorithmus in jedem Schritt mindestens eines der Sets vergrößert oder aber terminiert, ist die Anzahl der Iterationen der Hauptschleife durch $|V| \cdot |S|$ beschränkt. Zur heuristischen Beschleunigung des Algorithmus aktualisieren wir die Knoteninformationen nicht in beliebiger Reihenfolge wie in Algorithmus 7.1, sondern sortieren die Knoten initial in topologischer Reihenfolge mit dem Algorithmus von Kahn [Kahn, 1962], so daß jeder Knoten vor seinen Vorgängern im Kontrollflußgraphen an die Reihe kommt. Dies ist nur möglich wenn wir die Rücksprungkanten von Schleifen ignorieren. Die Idee hierzu findet sich bei Appel [Appel & Ginsburg, 1998].

Datenflußanalysen wurden in der Vergangenheit bereits intensiv erforscht. Viele weitere Resultate sind in der Literatur bereits verfügbar, wie z.B. der Beweis der Existenz eines Fixpunktes für Nicht-Bitvektor-Probleme (Knaster-Tarski Fixpunkt-Theorem, s. [Rüthing, 2008]) und der Beweis, daß der obige Algorithmus korrekt ist und für Bitvektorprobleme stets optimale Lösungen liefert (Koinzidenz-Theorem, s. [Knoop & Steffen, 1992]). Diese weiterführenden Resultate sollen hier aber nicht weiter vertieft werden.

In Abbildung 7.6 ist ein Beispiel für die Analyse der Lebensdauer von Symbolen in einer Funktion `foo` dargestellt, der entsprechende Quellcode ist in Abbildung 7.5 zu finden. An jedem Statement sind die LIVE-IN$_{\text{may}}$- (oberhalb) und LIVE-OUT$_{\text{may}}$-Mengen (unterhalb) angetragen, die sich nach Abschluß des Algorithmus ergeben, allerdings ist aus Platzgründen immer nur der erste Buchstabe jedes Symbolnamens angegeben. Der Startknoten der Funktion ist hell-, der Endknoten dunkelgrau markiert. Am `return`-Statement sind initial nur die globalen Symbole `buffer` und `intbuffer` sowie das durch das Pointer-Argument sichtbar gemachte `arg` lebendig. Beim Abarbeiten der initialen `worklist` würde z.B. am Knoten `while ( *value )` zuerst das LIVE-OUT$_{\text{may}}$-Set `{arg,buffer,result}` eingetragen, da anfangs das LIVE-IN$_{\text{may}}$-Set von `(*value)-;` noch leer ist. Erst bei der erneuten Bearbeitung des `while`-Knotens würde `value` in das LIVE-OUT$_{\text{may}}$-Set eingetragen, da dann auch das LIVE-IN$_{\text{may}}$-Set von `(*value)-;` berechnet worden ist. Die unterschiedliche Behandlung von skalaren Variablen und Arrays/Strukturen zeigt sich an den Knoten `intbuffer = result;` und `buffer.fa = result;`. Im ersten Fall ist `intbuffer` danach tot, im zweiten Fall ist `buffer` dies nicht, da über die lokalen Transferfunktionen nicht festgestellt werden kann, ob noch weitere Felder in der Struktur existieren, die noch nicht tot sind.

```
int foo( int* v ) {
  int r = 0;

  if ( v ) {
    b.fa = r;
    b.fb = *v;

    _Pragma( "loopbound min
                10 max 10" )
    while ( *v ) {
      ( *v )--;
    }
  } else {
    r = 1;
  }

  i = r;
  return r;
}
```

```
struct stype {
  int fa;
  int fb;
};

struct stype b =
  { 0, 0 };

int i = 0;


int main( void ) {
  int a = 10;
  foo( &a );
  return 0;
}
```

Abbildung 7.5.: Beispielcode für die Berechnung der Lebensdauer von Symbolen.

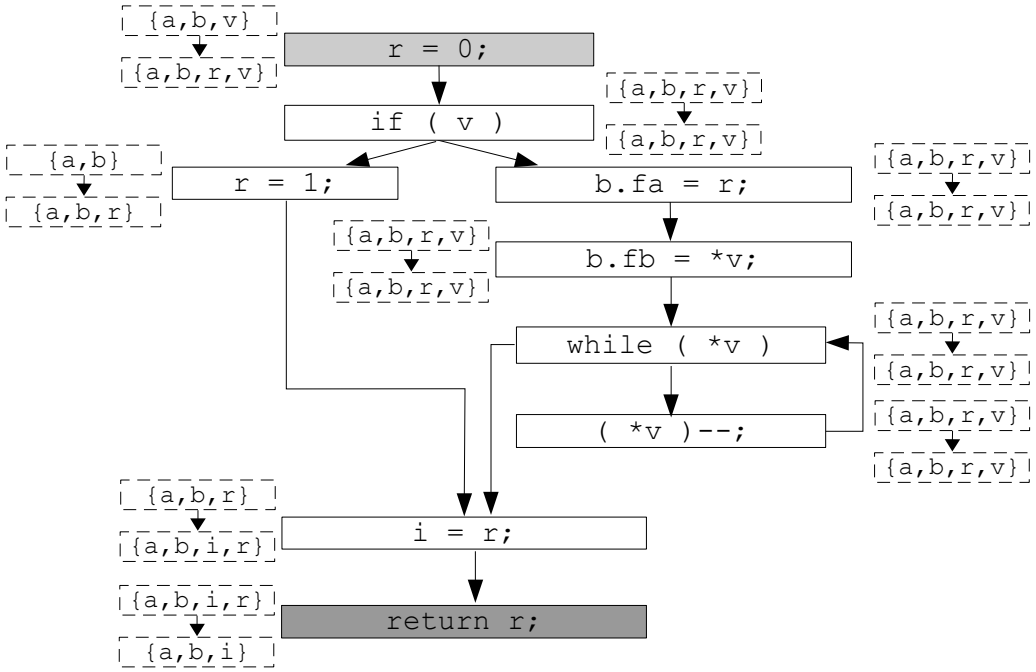


Abbildung 7.6.: Mit LIVE-IN$_{may}$ und LIVE-OUT$_{may}$ Mengen annotierter CFG des Codebeispiels .

7.2. Common Subexpression Elimination

Die ***Common Subexpression Elimination*** ist neben der ***Loop Invariant Code Motion*** und der ***Partial Redundancy Elimination*** eine der gebräuchlichsten Optimierungen zur Entfernung von redundanten Berechnungen aus Programmen. Von diesem drei Optimierungen werden wir jedoch nur die CSE als Superblock-spezifische Variante implementieren. Argumente gegen die Superblock-spezifische Variante der Loop Invariant Code Motion haben wir in Abschnitt 4.4 bereits aufgeführt. Gegen die Partial Redundancy Elimination spricht, daß sie keine direkte High-Level-Optimierung ist, und daß sie im WCC nicht direkt umsetzbar ist, da sie eine Zwischendarstellung des Programms benötigt, die der WCC bisher nicht enthält.

Wir folgen bei der Durchführung der CSE im Wesentlichen der Darstellung von Ghiya [Ghiya, 1998], der die CSE ebenfalls auf Basis von Def/Use-Sets durchführt. Der vollständige Algorithmus ist in Listing 7.3 dargestellt - durch die Zuhilfenahme der Def/Use-Sets ist eine relativ kompakte Formulierung möglich. Wir werden mit der CSE jeden Superblock einzeln und direkt nach seiner Erstellung optimieren (s. Abbildung 6.1). Dabei wird der Superblock vom Beginn bis zum Ende durchlaufen (Listing 7.3, Zeile 7/8) und an jedem neuen Statement wird die Menge der Ausdrücke aktualisiert die bis zu dieser Stelle im Superblock verfügbar ist. Wir nennen einen Ausdruck a dabei ***verfügbar*** an einer Anweisung s im Superblock, wenn sein Wert auf dem Superblock bereits in einer vorhergehenden Anweisung s_{comp} berechnet wurde und im Superblock-Code von s_{comp} bis einschließlich s keines der Symbole aus $\text{USE}_{\text{may}}(a)$ überschrieben wurde. Die aktuelle Menge der verfügbaren Ausdrücke wird dabei während der CSE in einer Menge `availSet` gespeichert, das bei jedem neuen Ausdruck `exp`, der während des Durchlaufens des Superblocks entdeckt wird, aktualisiert wird (Listing 7.3, Zeile 14-18):

1. Alle Kandidaten im `availSet`, die durch die Schreibzugriffe in `exp` nicht mehr verfügbar (***killed***) sind, werden aus dem `availSet` gelöscht. (Listing 7.3, Zeile 15-17)
2. Falls `exp` keine Variable sowohl liest als auch beschreibt, falls die Zwischenspeicherung von `exp` zulässig ist und `exp` selbst nicht ersetzt werden soll, so wird der Ausdruck in das `availSet` aufgenommen (Listing 7.3, Zeile 32-34).

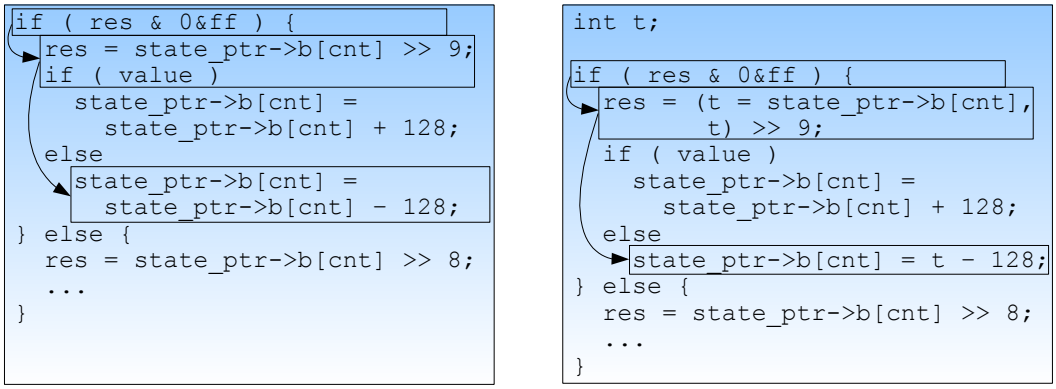
Ausdrücke `exp`, deren Zwischenspeicherung zulässig ist, werden als ***CSE-Kandidaten*** bezeichnet und müssen folgende Kriterien erfüllen:

- Es handelt sich um eine Auswertung eines globalen Symbols
- Es handelt sich um einen Ausdruck mit unären, binärem oder `?:`-Operator oder um einen Zugriff auf eine Struktur (`.`, `->`) oder ein Array (`[]`),
  - der keine Schreibzugriffe enthält ($\text{DEF}_{\text{may}} = \emptyset$)
  - der skalaren Typ hat, also keine Struktur und keinen Array zurückgibt
  - der nicht Teil einer `sizeof`-Berechnung ist
  - dessen Adresse nicht ermittelt werden soll (kein direktes Ziel des `&`-Operators)

Falls ein CSE-Kandidat `availExp` an einer Anweisung verfügbar ist und in ihr in einem identischen Ausdruck `exp` neu berechnet werden soll (Listing 7.3, Zeile 21-29), so können wir an dieser Stelle die Neuberechnung eliminieren. Dazu wird ein Eintrag `availExp` $\rightarrow$ `exp` in der Zuordnungstabelle `replacementMap` angelegt. Nachdem diese Einträge angelegt worden sind muß überprüft werden, ob zwischen dem aktuellen Basisblock und seinem Superblock-Nachfolger eine innere Schleife liegt (Listing 7.3, Zeile 41-51), da diese inneren Schleifen nicht explizit gespeichert werden (s. Abschnitt 6.3.2). Falls eine oder mehrere innere Schleifen gefunden werden, so müssen alle Ausdrücke innerhalb der Scheifen gescannt werden und diejenigen verfügbaren Ausdrücke, die durch die Schleifenanweisungen nicht länger verfügbar sind, werden aus dem `availSet` entfernt (Listing 7.3, Zeile

```
1  void superblockCSE( Superblock t )
2  {
3    set<IR_Exp*> availList;
4    multimap<IR_Exp*, IR_Exp*> replacementMap;
5
6    // Iteriere über den Superblock, vom Anfang bis zum Ende
7    IR_BasicBlock *curBlock = startNode( t );
8    while ( curBlock != endNode( t ) ) {
9
10     for ( IR_Stmt *stmt : curBlock->getStatementsInBBOrder() ) {
11       for ( IR_Exp *exp : stmt->getExpressionsInEvalOrder() ) {
12
13         // Aktualisiere die Menge verfügbarer Ausdrücke
14         for ( IR_Exp *availExp : availList ) {
15           if ( USE_may(availExp) ∩ DEF_may(exp) ≠ ∅ ) {
16             availList.erase( availExp );
17           }
18         }
19
20         if ( isCSECandidate( exp ) ) {
21           bool willBeReplaced = false;
22
23           // Merke den Ausdruck für die spätere Ersetzung vor
24           for ( IR_Exp *availExp : availList ) {
25             if ( *availExp == *exp ) {
26               replacementMap.insert( availExp → exp );
27               willBeReplaced = true;
28             }
29           }
30
31           // Oder liste ihn von hier an ggf. ebenfalls als verfügbar
32           if ( !willBeReplaced && USE_may(exp) ∩ DEF_may(exp) = ∅ ) {
33             availList.insert( exp );
34           }
35         }
36       }
37     }
38
39     // Entferne Ausdrücke, die durch Anweisungen in
40     // inneren Schleifen unverfügbar werden
41     if ( innerLoopAheadOf( curBlock ) ) {
42       for ( IR_Stmt *loopstmt : innerLoop->getStatements() ) {
43         for ( IR_Exp *loopexp : stmt->getExpressions() ) {
44           for ( IR_Exp *availExp : availList ) {
45             if ( USE_may(availExp) ∩ DEF_may(loopexp) ≠ ∅ ) {
46               availList.erase( availExp );
47             }
48           }
49         }
50       }
51     }
52
53     curBlock = traceSuccessorBlock( curBlock );
54   }
55
56   // Führe die Ersetzungen wie beschrieben durch
57   doReplacements( replacementMap );
58 }
```

Algorithmus 7.3: Algorithmus für die Superblock-CSE.



(a) Original-Code (Superblock ist markiert).

(b) Superblock-CSE wurde angewandt.

Abbildung 7.7.: Beispiel für die Anwendung der SB-CSE.

45-47). Erst danach kann die Abarbeitung des nächsten Basisblock auf dem Superblock erfolgen.

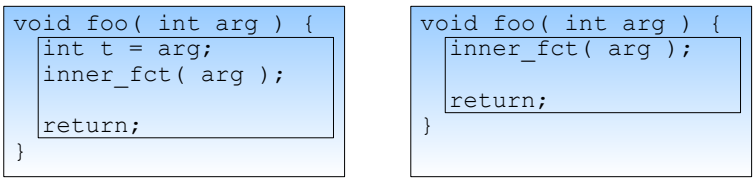
Sobald der Superblock vollständig durchlaufen wurde, wird die `replacementMap` in der Funktion `doReplacements` durchlaufen und alle dort vermerkten Ersetzungen werden durchgeführt (Listing 7.3, Zeile 57). Dabei wird für jeden Ausdruck `availExp`, zu dem eine Menge von redundanten Neuberechnungen $\texttt{exp}_1, ..., \texttt{exp}_n$ registriert wurde, folgendes durchgeführt:

- Wir erzeugen ein neues Symbol `t` mit dem Typ von `availExp` im Top-Level-Compound der Funktion.
- Wir ersetzen `availExp` durch den Ausdruck `(t = availExp, t)` und passen die WCET und Codegrößen-Annotationen des umgebenden Blocks über unsere Schätzheuristik an.
- Wir ersetzen alle $\texttt{exp}_1, ..., \texttt{exp}_n$ durch eine Auswertung von `t` und passen auch dort jeweils die WCET- und Codegrößenwerte an.

Ein Beispiel für die Anwendung der CSE ist in Abbildung 7.7 dargestellt. Hierbei handelt es sich um ein leicht verkürztes Stück Code aus einem G.721-Encoder aus der Testbench des WCC. `value` und `res` seien hierbei globale Variablen und `state_ptr` sei ein Pointer auf eine Struktur mit einem Feld `b`. Die einzelnen Zustände der `availSet` beim Durchlaufen des Superblocks sind:

1. `{res, res & 0&ff}` (nach `if( res & 0&ff )`)
2. `{state_ptr->b, state_ptr->b[cnt], state_ptr->b[cnt] » 9}`
   (nach der Zuweisung an `res`)
3. `{state_ptr->b, state_ptr->b[cnt], state_ptr->b[cnt] » 9, value}`
   (nach `if( value )`)
4. `{value}` (nach der Zuweisung an `state_ptr->b[cnt]`)

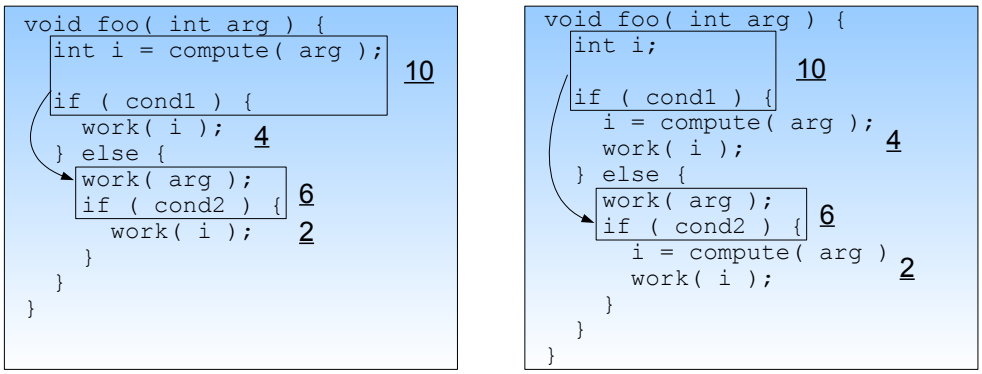
Im letzten Schritt werden keine neuen verfügbaren Ausdrücke generiert, weil dort durch die Zuweisung `state_ptr->b[cnt]=state_ptr->b[cnt]-128` alle gelesenen Ausdrücke direkt wieder "killed" werden. Außerdem ist nur an dieser Anweisung ein Ausdruck verfügbar, der dort auch gelesen wird (`state_ptr->b[cnt]`), so daß dieser Ausdruck dort in das `replacementSet` eingetragen wird, und nach dem Durchlaufen des Superblocks ersetzt wird.



(a) Original-Code (Trace ist markiert).

(b) Toter Code wurde eliminiert.

Abbildung 7.8.: Beispiel für die Anwendung der SB-DCE bei totem Code.



(a) Original-Code (Trace ist markiert).

(b) Superblock-toter Code wurde eliminiert.

Abbildung 7.9.: Beispiel für die Anwendung der SB-DCE bei Superblock-totem Code.

7.3. Dead Code Elimination / Operation Migration

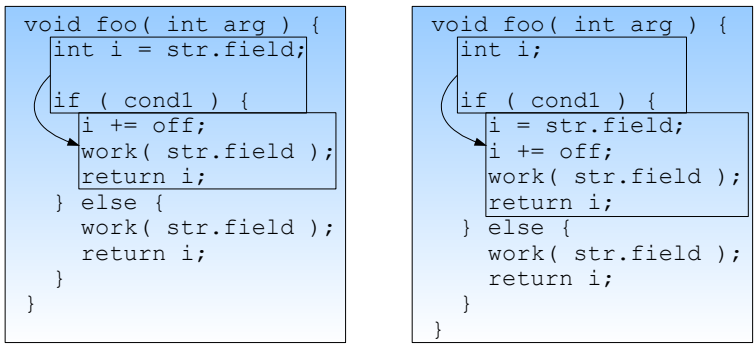
Im Gegensatz zur Superblock-CSE versucht die Superblock-DCE nicht nur redundante Berechnungen einzusparen sondern Anweisungen vollständig von Superblöcken zu löschen, oder, falls dies nicht möglich ist, die Anweisung vom Superblock weg in die Verzweigungen zu verschieben, in denen sie benötigt werden. Dazu teilen wir die Anweisungen auf dem Superblock in drei Kategorien ein:

- ***Tote*** Anweisungen s_{dead} sind Anweisungen mit $\text{LIVE-OUT}_{\text{may}}(s_{dead}) \cap \text{DEF}_{\text{may}}(s_{dead}) = \emptyset$. Die Resultate, die von ihnen berechnet werden, werden an keiner Stelle im Programm mehr benutzt.

- ***Superblock-tote*** Anweisungen s_{sbdead} sind Anweisungen, die nicht tot sind und bei denen alle Variablen aus $\text{DEF}_{\text{may}}(s_{sbdead})$ auf dem Superblock nicht gelesen werden bevor sie überschrieben werden. Die Resultate, die von ihnen berechnet werden, werden an keiner Stelle im Superblock mehr benutzt, dafür aber außerhalb des Superblocks.

- ***Lebendige*** Anweisungen sind Anweisungen, die weder tot noch Superblock-tot sind.

Tote Anweisungen können ersatzlos gestrichen werden und tragen somit nicht mehr zur Laufzeit des WCEP bei. Beispielcode hierzu, wie er z.B. nach einem Function Inlining und anschließender Value Propagation entstehen kann, findet sich in Abbildung 7.8. In Abildung 7.8(b) ist die tote Zuweisung an `t` eliminiert worden. Superblock-tote Anweisungen s_{sbdead} können aus einem Superblock $T = (b_1, ..., b_n)$ heraus verschoben werden. Dabei wird eine Kopie von s_{sbdead} an jedem Superblock-Aussprungpunkt b_{out} mit $\exists i \in \{1, ...n\} : (b_i, b_{out}) \in E \wedge b_{out} \neq b_{i+1}$ angelegt, an dem



(a) Original-Code (Trace ist markiert).

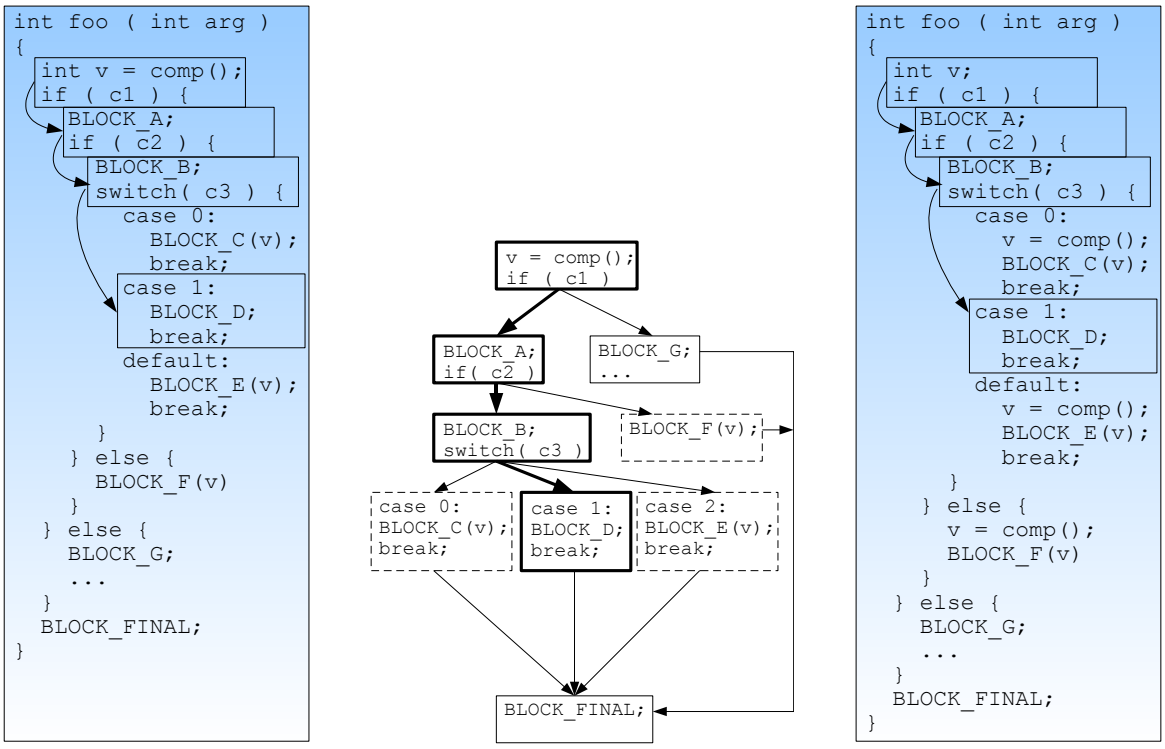
(b) Nach dem Code Sinking.

Abbildung 7.10.: Beispiel für die Anwendung des Code Sinking bei lebendigem Code.

$\text{LIVE-IN}_{\text{may}}(b_{out}) \cap \text{DEF}_{\text{may}}(s_{sbdead})$ gilt. Da alle Aussprungpunkte eine Ausführungshäufigkeit besitzen die kleiner oder gleich der des Superblocks ist, wird hierdurch die Ausführungshäufigkeit des verschobenen Statements nicht vergrößert und im günstigen Fall verringert. Ein Beispiel hierzu findet sich in Abbildung 7.9, die Block-WCECs sind dort jeweils unterstrichen dargestellt. Lebendige Anweisungen können höchstens innerhalb des Superblocks an die Stelle verschoben werden, an der sie benötigt werden, was auch als ***Code Sinking*** bezeichnet wird. Die Hoffnung ist hierbei, daß dort nach der Verschiebung andere Optimierungen angewandt werden können, um den Code weiter zu vereinfachen. Im Beispiel aus Abbildung 7.10 kann hierdurch z.B. die doppelte Berechnung für den Zugriff auf das Feld `field` in der globalen Struktur `str` vermieden werden, indem entweder die High-Level-CSE oder die Low-Level-Optimierungen die doppelte Berechnung der entsprechenden Adresse erkennen und eliminieren. Die Superblock-DCE wurde erstmals von Chang in [Chang et al., 1991] erwähnt, die Idee des Code Sinkings wurde von Zhao in [Zhao et al., 2006] im Zusammenhang mit der Superblockoptimierung erwähnt.

Die Grundidee der Superblock-DCE ist, daß wir den Superblock durchlaufen und dabei das bearbeitete Statement s an jeder Verzweigung auf dem Superblock in alle Nicht-Superblock-Nachfolger b_{succ} kopieren, an denen das Statement lebendig ist ($\text{LIVE-IN}_{\text{may}}(b_{succ}) \cap \text{DEF}_{\text{may}}(s) \neq \emptyset$). Wir stoppen das Durchlaufen des Superblocks, falls wir auf ein Statement treffen, das die weitere Verschiebung blockiert (Code Sinking) oder falls das Superblock-Ende erreicht ist. Falls das bearbeitete Statement s nach dem Superblock-Endblock b_{end} noch lebendig ist ($\text{LIVE-OUT}_{\text{may}}(b_{end}) \cap \text{DEF}_{\text{may}}(s) \neq \emptyset$) müssen wir eine Kopie von s am Ende des Superblocks einfügen. Das Original-Statement können wir danach in jedem Fall löschen. Ein Beispiel ist in Abbildung 7.11 dargestellt. Alle Blöcke, die `v` lesen, sind mit `(v)` markiert. Bei der Dead Code Elimination würde der Superblock, wie in Abbildung 7.11(b) zu sehen, vom Anfang (oben) bis zum Ende (unten) durchlaufen, und in jedem gestrichelt gekennzeichneten Block würde eine Kopie von `v = comp()` hinterlassen. Das Original-Statement würde hier am Ende gelöscht, da es im nachfolgenden Code `BLOCK_FINAL` nicht mehr lebendig ist.

Der Rahmenalgorithmus für die Superblock Dead Code Elimination ist in Listing 7.5 dargestellt, die verwendeten Datenstrukturen sind aus Platzgründen gesondert in Listing 7.4 zu finden. Wir iterieren wieder über den gesamten Superblock, vom Anfang bis zum Ende (Listing 7.5, Zeilen 10-44), und sammeln dabei die Statements auf, die Kandidaten für die DCE sind (Listing 7.5, Zeilen 16-19). ***DCE-Kandidaten*** sind dabei nur Statements vom Typ `IR_ExpStmt`, die keine Aufrufe von externen Funktionen (deren Quellcode unbekannt ist) und keine Benutzungen von `volatile`-



(a) Original-Code (Trace ist markiert).

(b) Kontrollflußgraph des Original-Codes (Trace ist markiert).

(c) Nach der Superblock-DCE.

Abbildung 7.11.: Beispiel für Anwendung der SB-DCE.

```
1  // Potentiell bewegliche Kandidaten (tot / sb-tot / lebendig)
2  set<IR_Stmt*> dceCandidates;
3  // Anweisungen die definitiv verschoben werden
4  set<IR_Stmt*> dceStmtsToMove;
5  struct BlockageInfo {
6    IR_Stmt* blockStmt;
7    enum InsertDirection { IN_FRONT_OF_STMT, BEHIND_STMT } stopSide;
8  }
9  // Für jedes Stmt in dceStmtsToMove: Ist seine Bewegung blockiert?
10 map<IR_Stmt*, struct BlockageInfo> candMovePosition;
11 // Anweisungen deren Bewegung entlang des Superblock blockiert ist
12 set<IR_Stmt*> notMovable;
13 // Menge der Symbole die ein Kandidat definiert,
14 // und die noch lebendig sein könnten
15 map<IR_Stmt*, set<IR_Symbol*> *> stillLiveDefs;
16 // Beim Durchlaufen von Schleifen: Letztes Stmt vor der Schleife
17 IR_Stmt *lastOnTraceStmt;
```

Algorithmus 7.4: Datenstrukturen für die Superblock-DCE.

```
1  void superblockDCE( Superblock t )
2  {
3    do {
4      // Datenstrukturen initialisieren
5      dceCandidates.clear(); dceDeadStmts.clear(); candMovePosition.clear();
6      notMovable.clear(); stillLiveDefs.clear(); lastOnTraceStmt = 0;
7
8      // Iteriere über den Superblock, vom Anfang bis zum Ende
9      IR_BasicBlock *curBlock = startNode( t );
10     while ( curBlock != endNode( t ) ) {
11
12       // Kandidateninformationen aktualisieren
13       for ( IR_Stmt *newStmt : curBlock->getStatements() ) {
14         filterCandidates( newStmt, 0 );
15
16         if ( isDCECandidate( newStmt ) ) {
17           dceCandidates.insert( newStmt );
18           stillLiveDefs[ newStmt ] = DEF_may(stmt);
19         }
20       }
21
22       // Behandle innere Schleifen bis zum evtl. vorh. Aussprungpunkt
23       for ( IR_LoopStmt *loop : innerLoopsAheadOf( curBlock ) ) {
24         if ( parentCompound( loop ) == parentCompound( curBlock ) ) {
25           for ( IR_Stmt *lstmt : topSort( loop->getStatements() ) ) {
26             filterCandidates( lstmt, lastStmtOf( curBlock ) );
27           }
28         }
29       }
30
31       // Unbewegliche Kandidaten löschen / stoppen
32       checkImmovableCandidates( curBlock );
33
34       // Behandle innere Schleifen nach dem evtl. vorh. Aussprungpunkt
35       for ( IR_LoopStmt *loop : innerLoopsAheadOf( curBlock ) ) {
36         if ( parentCompound( loop ) != parentCompound( curBlock ) ) {
37           for ( IR_Stmt *lstmt : topSort( loop->getStatements() ) ) {
38             filterCandidates( lstmt, lastStmtOf( curBlock ) );
39           }
40         }
41       }
42
43       curBlock = traceSuccessorBlock( curBlock );
44     }
45
46     // Alle verbleibenden Kandidaten sind tot oder Superblock-tot
47     for ( IR_Stmt *cand : dceCandidates ) {
48       if ( cand ∈ notMovable ) {
49         insertStmtToMove( cand, candMovePosition[ cand ].blockStmt,
50           candMovePosition[ cand ].stopSide );
51       } else {
52         insertStmtToMove( cand, 0, IN_FRONT_OF_STMT );
53       }
54     }
55
56   // Solange neuer toter Code gefunden wurde: Suche erneut
57   } while ( performCodeMovement() );
58 }
```

Algorithmus 7.5: Rahmenalgorithmus für die Superblock-DCE.

Symbolen enthalten. Diese Kandidaten-Anweisungen (`newStmt`) werden zuerst als "potentiell tot / sb-tot" in die Menge `dceCandidates` aufgenommen (Listing 7.5, Zeile 17) und die Menge der in ihnen enthaltenen, potentiell lebendigen Definitionen wird mit DEF_{may}(`newStmt`) initialisert (Listing 7.5, Zeile 18). Außerdem wird an jedem neuen Statement, das auf dem Superblock liegt, die Menge der bisher eingetragenen Kandidaten aktualisiert (Listing 7.5, Zeile 14). Falls sich z.B. herausstellt, daß ein Kandidat nicht tot sein kann, weil seine definierten Symbole auf dem Superblock wieder gelesen werden, so wird dieser Kandidat aus `dceCandidates` entfernt. Die hierfür benutzte Funktion `filterCandidates` werden wir weiter unten im Detail besprechen. Dieselbe Funktion wenden wir danach in den Zeilen 23-29 (Listing 7.5) in der Reihenfolge ihres Auftretens im Code auch auf alle Statements innerhalb von inneren Schleifen an. Dabei werden vorerst nur solche inneren Schleifen betrachtet, die *vor* einem möglichen Aussprungpunkt aus dem Superblock liegen (Zeile 24). Die Funktion `topSort` sortiert die Schleifenanweisungen topologisch (ohne Beachtung von Rücksprungkanten) und stellt somit sicher, daß wir die Anweisungen in der Reihenfolge ihres Auftretens im Schleifenrumpf durchlaufen. Dies ist nötig, da die `filterCandidates`-Funktion die Statements in der Reihenfolge ihrer Abarbeitung erwartet. Nach dem dieser Teil der inneren Schleifen durchlaufen wurde, wird die Funktion `checkImmovableCandidates` aufgerufen (Listing 7.5, Zeile 32), die in Listing 7.8 aufgeführt ist. Dazu soll hier schon erwähnt werden, daß die Funktion `filterCandidates` Datenabhängigkeiten zwischen Statements erkennt und in `candMovePosition` und `notMovable` einträgt. `checkImmovableCandidates` überprüft für die Statements, die nicht über den aktuellen Block hinweg verschoben werden können (diese wurden von `filterCandidates` in die Menge `notMovable` eingetragen), ob sie in eine Aussprungkante oder hinter das Superblock-Ende kopiert werden müssten, um die Korrektheit des Programms zu erhalten (Listing 7.8, Zeile 5-8). Falls das der Fall ist, werden diese Statement aus der Kandidatenmenge entfernt (Listing 7.8, Zeile 18/19), oder aber, falls das Code Sinking aktiviert und anwendbar ist, bis vor das sie blockierende Statement bewegt (Listing 7.8, Zeile 13/14). Nach dem Aufruf von `checkImmovableCandidates` bleiben also nur noch die Kandidaten erhalten, die entlang des Superblock weiterverschoben werden können. Diese werden dann durch Scannen der verbleibenden inneren Schleifen gefiltert (Listing 7.5, Zeile 35-41), bevor der Algorithmus zum nächsten Basisblock auf dem Superblock wechselt (Listing 7.5, Zeile 43). Wenn der komplette Superblock durchlaufen wurde, sind alle verbleibenden Kandidaten in `dceCandidates` entweder tot oder Superblock-tot. In Zeile 47-54 von Listing 7.5 wird diese Information für die `performCodeMovement`-Funktion abgelegt, wobei die Statements, deren Bewegung auf dem Superblock blockiert ist, nur bis zum blockierenden Statement bewegt werden können. Die in Zeile 49 und 52 benutzte `insertStmtToMove`-Funktion ist in Listing 7.8 aufgeführt. In Zeile 57 wird `performCodeMovement` aufgerufen und führt die geplanten Codebewegungen durch. Die gesamte Prozedur wird solange wiederholt, wie hierbei noch Codebewegungen erzielt werden können.

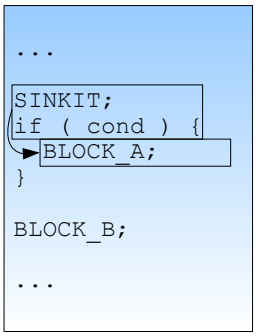
Die Funktion `filterCandidates` ist in Listing 7.6 beschrieben. Der Parameter `newStmt` ist dabei das jeweils nächste Statement auf dem Superblock oder innerhalb einer inneren Schleife auf dem Superblock. Falls `newStmt` eine Anweisung aus einer inneren Schleife ist, können wir dies über den Parameter `lastOnTraceStmt` erkennen, der in diesem Fall auf das letzte Statement auf dem Superblock zeigt und sonst gleich 0 ist. `filterCandidates` iteriert über alle registrierten DCE-Kandidaten (Zeile 3) und erfüllt dabei zwei verschiedene Aufgaben:

- Ausfiltern von lebendigen Anweisungen
- Erkennung und Registrierung von Blockierzuständen z.B. aufgrund von Datenabhängigkeiten

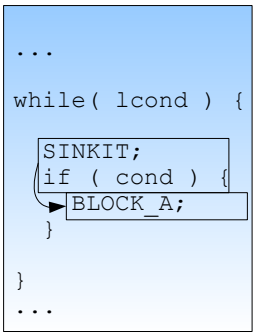
Der erste Punkt wird in den Zeilen 4 bis 20 abgedeckt, der zweite in den Zeilen 28 bis 57. Zuerst wird überprüft, ob das neue Statement `newStmt` eine der vom Kandidaten definierten Variablen ausliest (Zeile 4). In diesem Fall ist der Kandidat lebendig und wird aus `dceCandidates` entfernt. Falls aktiviert (Zeile 16), ist es an der Stelle höchsten noch möglich das Code Sinking durchzuführen. Dazu muß erst festgestellt werden, bis wohin wir den Kandidaten verschieben könnten (Zeile

```
1  void filterCandidates( IR_Stmt &newStmt. IR_Stmt *lastOnTraceStmt )
2  {
3    for ( IR_Stmt *cand : dceCandidates ) {
4      if ( USE_may(newStmt) ∩ stillLiveDefs[ cand ] ≠ ∅ ) {
5        dceCandidates.erase( cand );
6
7        // Kandidat ist nicht tot -> Instruction Sinking?
8        struct BlockageInfo binfo;
9        if ( cand ∈ notMovable ) {
10         binfo = candMovePosition[ cand ];
11       } else if ( lastOnTraceStmt ) {
12         binfo = { lastOnTraceStmt, BEHIND_STMT };
13       } else {
14         binfo = { newStmt, IN_FRONT_OF_STMT };
15       }
16       if ( configuration.doInstructionSinking() &&
17            binfo.blockStmt->getBasicBlock() != cand->getBasicBlock() ) {
18         insertStmtToMove( cand, binfo.blockStmt, binfo.stopSide );
19       }
20     } else {
21       // Verbleibende lebendige Symbole aktualisieren
22       if ( !lastOnTraceStmt || ( newStmt.getBasicBlock() pdom
23                                  lastOnTraceStmt->getBasicBlock() ) ) {
24         stillLiveDefs[ cand ]->erase( filterAS( DEF_must(newStmt) ) );
25       }
26
27       // Datenabhängigkeiten prüfen -> Kandidat kann blockiert werden
28       if ( DEF_may(newStmt) ∩ USE_may(cand) ≠ ∅ ||
29            DEF_may(newStmt) ∩ DEF_may(cand) ≠ ∅ ) {
30         notMovable.insert( cand );
31         if ( lastOnTraceStmt ) {
32           candMovePosition[ cand ].blockStmt = lastOnTraceStmt;
33           candMovePosition[ cand ].stopSide  = BEHIND_STMT;
34         } else {
35           candMovePosition[ cand ].blockStmt = newStmt;
36           candMovePosition[ cand ].stopSide  = IN_FRONT_OF_STMT;
37         }
38       }
39
40       // Spezialfälle, in denen keine Kopieroperationen zu Nachfolgern möglich sind
41       if ( !lastOnTraceStmt ) {
42         IR_BasicBlock *curBlock = newStmt.getBasicBlock();
43         for ( IR_BasicBlock *succBlock : δ+(curBlock) ) {
44
45           bool isPostdominatedByOutsider = succBlock pdom cand->getBasicBlock() &&
46             traceIndex( succBlock ) != traceIndex( succBlock );
47           bool isBackEdge = isBackedge( curBlock, succBlock );
48
49           if ( ( isPostdominatedByOutsider || isBackEdge ) &&
50                needsToMoveOverBranch( cand, succBlock ) ) {
51             notMovable.insert( cand );
52             candMovePosition[ cand ].blockStmt = newStmt;
53             candMovePosition[ cand ].stopSide  = IN_FRONT_OF_STMT;
54             break;
55           }
56         }
57       }
58     }
59   }
60 }
```

Algorithmus 7.6: Hilfsfunktion `filterCandidates` für die Superblock-DCE.



(a) Ausnahme: Nachfolger postdominiert Superblockknoten.



(b) Ausnahme: Nachfolger ist Schleifenkopf (Rückkante).

Abbildung 7.12.: Ausnahmesituationen bei der SB-DCE.

8-15): Entweder bis vor das blockierende Statement (Zeile 10), bis vor die aktuell gescannte innere Schleife (Zeile 12) oder bis vor das `newStmt` (Zeile 14). Wenn das Ziel der Codebewegung nicht im selben Block liegt, wie der Kandidat selbst (Zeile 17), wird der Kandidat für die Codebewegung registriert (Zeile 18).
Wenn der Kandidat nicht als lebendig eingestuft wird, werden die verbleibenden lebendigen Symbole des Kandidaten auf dem Superblock ermittelt. Hierzu werden alle Symbole, die in `newStmt` definitv überschrieben werden (DEF_{must}(`newStmt`)) aus dem `stillLiveDefs`-Set des Kandidaten entfernen (Zeile 24). Allerdings müssen wir hier, wie auch schon bei der Berechnung der Lebensdauer-Informationen, mit der `filterAS`-Funktion alle Array- und Struktur-Symbole ignorieren, da wir nicht sicherstellen können, daß diese vollständig überschrieben worden sind. Außerdem müssen wir noch den Spezialfall betrachten, daß `newStmt` Teil einer inneren Schleifen sein kann. In diesem Fall dürfen wir das `stillLiveDefs`-Set nur aktualisieren, falls `newStmt` in jedem möglichen Ausführungspfad ausgeführt wird, falls es also das letzte Statement vor der Schleife postdominiert (Zeile 22-23).
Im Rest des Codes (Zeile 28-57) werden die Datenabhängigkeiten überprüft und zwei Spezialfälle abgefangen. Eine Datenabhängigkeit besteht, wenn `newStmt` eines der Symbole, die `cand` liest oder schreibt, überschreibt (Zeile 28-29). In dem Fall darf `cand` nicht über `newStmt` hinweg bewegt werden (Zeile 35-36), bzw. nicht über die Schleife hinweg, falls wir uns in einer inneren Schleife befinden (Zeile 32-33). Der Code in den Zeilen 41-57 deckt zwei Sonderfälle ab, die beispielhaft in Abbildung 7.12 dargestellt sind. Wir können die Statements nur über eine Verzweigung auf dem Superblock hinweg bewegen, wenn wir sicher stellen können, daß das verschobene Statement dabei in jedem der Verzweigungsziele, in denen es lebendig ist, weiterhin exakt einmal ausgeführt wird. Dies ist bei allen Verzweigungen, die keine Möglichkeit bieten die Verzweigung zu überspringen, gegeben. Die Superblockbildung fügt `default`-Statements in alle `switch`es ein und konvertiert `if`-Anweisungen zu `if-else`-Anweisungen, wodurch diese Eigenschaft für fast alle Verzweigungen erfüllt ist. Es gibt nur eine Situation, in der die Superblockbildung ein `if` auf dem Superblock nicht zu einem `if-else` konvertiert, nämlich genau dann, wenn das Ende des Superblocks innerhalb des `if`s liegt, wie in Abbildung 7.12 jeweils angedeutet. In diesen Fällen können wir Statements, die nach dem `if` noch lebendig sind (in den Beispielen soll `SINKIT` nach den `if`s jeweils noch lebendig sein), *nicht* einfach vor den Nachfolgerknoten (`BLOCK_B` in Abbildung 7.12(a)) kopieren, da sie sonst auf einem der Pfade doppelt ausgeführt würden (Abbildung 7.12(a) bzw. Zeile 45-46) oder da der Nachfolger ein Schleifenkopf ist (Abbildung 7.12(b) bzw. Zeile 47). Dieses Problem ließe sich durch Konvertierung des `if` zu einem `if-else` beheben, wir entscheiden uns hier jedoch dafür, solche Statements als blockiert zu betrachten, d.h. falls wir eine solche Situation entdecken (Zeile

45-50), so vermerken wir in den Blockierungsinformationen, daß der betroffene Kandidat nicht über den aktuell gescannten Basisblock hinaus bewegt werden darf (Zeile 51-54).

Nachdem der Superblock vollständig durchlaufen worden ist und alle zu verschiebenden Statements sowie deren Blockierinformationen ermittelt worden sind, führt `performCodeMovement` die geplanten Transformationen durch (Listing 7.5, Zeile 57). Der Quellcode zu dieser Funktion ist in Listing 7.7 aufgeführt. Auch hier wird wieder jedes zu verschiebende Statement `toMove` einzeln bearbeitet (Zeile 6). In Zeile 7 werden zuerst die evtl. an das Statement angehängten sowie die das Statement referenzierenden Flowfacts angepasst, so daß das Statement danach entfernt werden kann ohne die Flowfacts in einem ungültigen Zustand zurückzulassen. Dies ist hier im Gegensatz zur Situation bei der CSE notwendig, da wir die Anweisungen selbst verändern und teilweise löschen, während wir bei der CSE nur die in ihnen enthaltenen Ausdrücke modifiziert haben. Da die Flowfacts mit den Anweisungs-Objekten verknüpft sind, ist erst bei der Modifikation der Anweisungen ein Update der Flowfacts nötig. In Zeile 9 bis 11 werden Kontrollvariablen initialisiert, `stopMovingAt` ist das Statement an dem die Verschiebung enden muß, falls so ein Statement existiert, `curBlock` ist der aktuelle Basisblock und `traceSuccessor` ist sein Superblock-Nachfolger. Danach wird der Superblock bis zum Ende durchlaufen oder bis das Statement `stopMovingAt` erreicht ist (Zeile 14/15) und während dieses Durchlaufes wird `toMove` an Verzweigungen in jeden Nicht-Superblock-Nachfolger kopiert, an dem `toMove` lebendig ist (Zeile 22-29). Nach dem Superblock-Durchlauf wird in Zeile 36-43 `toMove` an die vorgemerkte Position neben das blockierende Statement kopiert. Falls kein blockierendes Statement existierte wird `toMove` wenn nötig in Zeile 47-51 hinter das letzte Superblock-Statement kopiert. Dies stellt jedoch einen ungünstigen und nicht wünschenswerten Fall dar, da somit der Superblock durch die komplette Verschiebung effektiv nicht verkürzt wird. Abschließend wird das Original-Statement gelöscht (Zeile 55/56) und der Optimierungszähler erhöht (Zeile 58). Die in diesem Abschnitt verwendeten Funktionen `annotationUpdateStmtAdded` und `annotationUpdateStmtRemoved` aktualisieren die `IR_WCETObject-`, `IR_CodeSizeObject-` und `IR_SuperblockInfo-`Annotationen, die an den betroffenen Statements angehängt sind, bzw. die auf diese verweisen. Die neuen WCET- bzw Codegrößen-Werte werden dabei wie immer mit den Heuristiken aus Abschnitt 5.2 aktualisiert. Nach dem Abschluß der `performCodeMovement`-Funktion wird eine neue Iteration der Hauptschleife eingeleitet (Listing 7.5), falls in dieser Runde noch toter Code gefunden wurde. Die DCE terminiert erst dann, wenn keine weiteren Codebewegungen mehr durchgeführt werden konnten.

```
1  unsigned int performCodeMovement()
2  {
3    unsigned int moveCount = 0;
4
5    // Führe die Codetransformation einzeln für jedes zu verschiebende Stmt aus
6    for ( IR_Stmt *toMove : dceStmtsToMove ) {
7      flowfactsUpdate( toMove );
8
9      IR_Stmt *stopMovingAt = candMovePosition[ toMove ].blockStmt;
10     IR_BasicBlock *curBlock = toMove->getBasicBlock();
11     IR_BasicBlock *traceSuccessor = 0;
12
13     // Behandle alle Aussprungpunkte bis zum Trace-Ende / dem blockierenden Statement
14     while ( traceSuccessor = tracePredecessorBlock( curBlock ) &&
15             stopMovingAt ∉ curBlock ) {
16
17       IR_BasicBlock *traceSuccessor = 0;
18       set<IR_BasicBlock*> *succs = 0;
19       if ( isBranchPoint( *curBlock, &succs, &traceSuccessor ) ) {
20
21         // Kopiere 'toMove' zum Aussprungziel falls nötig
22         for ( IR_BasicBlock *succBlock : succs ) {
23           if ( succBlock != traceSuccessor &&
24                needsToMoveOverBranch( toMove, succBlock ) ) {
25             IR_Stmt *copy = toMove->copy();
26             succBlock->insertStmtBefore( copy, firstStmtOf( succBlock ) );
27             annotationUpdateStmtAdded( copy );
28           }
29         }
30       }
31       curBlock = traceSuccessor;
32     }
33
34     if ( stopMovingAt ∈ curBlock ) {
35       // Superblock-Durchlauf wurde durch blockierendes Statement abgebrochen
36       IR_Stmt *copy = toMove->copy();
37       switch ( candMovePosition[ *itMove ].second ) {
38         case IN_FRONT_OF_STMT:
39           curBlock->insertStmtBefore( copy, stopMovingAt ); break;
40         case BEHIND_STMT:
41           curBlock->insertStmtBehind( copy, stopMovingAt ); break;
42       }
43       annotationUpdateStmtAdded( copy );
44
45     } else {
46       // Superblock-Durchlauf erreichte das Superblock-Ende
47       if ( needsToMoveOverSuperblockEnd( toMove, curBlock ) ) {
48         IR_Stmt *copy = toMove->copy();
49         curBlock->insertStmtBehind( copy, lastStmtOf( curBlock ) );
50         annotationUpdateStmtAdded( copy );
51       }
52     }
53
54     // Entferne das Original-Statement
55     annotationUpdateStmtRemoved( toMove );
56     toMove->getBasicBlock()->removeStmt( toMove );
57
58     moveCount++;
59   }
60   return moveCount;
61 }
```

Algorithmus 7.7: Hilfsfunktion `performCodeMovement` für die Superblock-DCE.

```
1  void checkImmovableCandidates( IR_BasicBlock *curBlock )
2  {
3    for ( IR_Stmt *cand : notMovable ) {
4      // Ermittle, ob der Kandidat aus dem Superblock heraus kopiert werden müsste
5      if ( ( isBranchPoint( curBlock ) &&
6             ∃offtraceBranch: needsToMoveOverBranch( cand, offtraceBranch ) ) ||
7           ( isSuperblockEnd( curBlock ) &&
8             needsToMoveOverSuperblockEnd( cand, endNode( t ) ) ) ) {
9        // Mögliches Instructions Sinking
10       if ( configuration.doInstructionSinking() &&
11            candMovePosition[ cand ].blockStmt->getBasicBlock()
12             != cand->getBasicBlock() ) {
13         insertStmtToMove( cand, candMovePosition[ cand ].blockStmt,
14           candMovePosition[ cand ].stopSide );
15       }
16
17       // Löschung des Kandidaten
18       dceCandidates.erase( cand );
19       notMovable.erase( cand );
20     }
21   }
22 }
23
24 bool needsToMoveOverBranch( IR_Stmt *candidate,
25                             IR_BasicBlock *branchTarget )
26 {
27   return DEF_may(candidate) ∩ LIVE-IN_may(firstStmtOf(branchTarget)) ≠ ∅;
28 }
29
30 bool needsToMoveOverSuperblockEnd( IR_Stmt *candidate,
31                                    IR_BasicBlock *superblockEnd )
32 {
33   return DEF_may(candidate) ∩ LIVE-OUT_may(superblockEnd) ≠ ∅;
34 }
35
36 void insertStmtToMove( IR_Stmt *toMove,
37   IR_Stmt* cBlockStmt, enum InsertDirection cStopSide )
38 {
39   bool blockingStmtWillBeMoved = cBlockStmt ∈ dceStmtsToMove;
40   bool moveStmtIsBlockingStmt = false;
41   for ( struct BlockageInfo binfo : candMovePosition ) {
42     if ( binfo.blockStmt == toMove ) {
43       moveStmtIsBlockingStmt = true; break;
44     }
45   }
46
47   // Statements dürfen nie in der selben Runde blockierend
48   // wirken und gleichzeitig selbst bewegt werden.
49   if ( !blockingStmtWillBeMoved && !moveStmtIsBlockingStmt ) {
50     dceStmtsToMove.insert( toMove );
51     candMovePosition[ toMove ].blockStmt = cBlockStmt;
52     candMovePosition[ toMove ].stopSide  = cStopSide;
53   }
54 }
```

Algorithmus 7.8: Hilfsfunktionen für die Superblock-DCE.

8. Auswertung

Um die erstellten Optimierungen zu evaluieren, wurden umfangreiche Tests durchgeführt, deren Ergebnisse in diesem Kapitel zusammengefasst sind. Einerseits dienten diese Tests dazu, die Korrektheit der implementierten Optimierungen sicherzustellen (Korrektheitstests, Abschnitt 8.1), und andererseits dazu, die Optimierungen durch Messung der erzielten WCET-, ACET- und Codegrößen-Ergebnisse zu bewerten (Leistungstests, Abschnitt 8.2).

8.1. Korrektheitstests

Um sicherzustellen, daß die implementierten Optimierungen korrekt arbeiten, wurden die Superblockbildung mit `else-if`-Neufaltung und `switch`-Konvertierung, die Superblock-CSE und die Superblock-DCE jeweils anhand der Testbench des WCC überprüft. Im WCC existiert dazu bereits eine Vielzahl an Benchmarks (siehe Anhang A), die jeweils mit Ausgabekommandos (`printf`) durchsetzt sind. Die Ausgabekommandos geben der Werte von internen Variablen des Benchmarks und insbesondere die Ergebnisse, die die Benchmark berechnet, aus. Falls es durch eine Optimierung zu einer nicht Semantik-erhaltenden Codetransformation kommt, so besteht eine sehr große Wahrscheinlichkeit, daß durch diese Transformation auch in mindestens einem der Benchmarks die Ausgaben verfälscht werden, da die Benchmarks ein breites Spektrum an syntaktischen und algorithmischen Konstruktionen abdecken. Das Testframework überprüft daher, ob die Ausgaben bei einer Referenzkompilierung und bei der Kompilierung mit eingeschalteter Superblock-Bildung, -CSE und -DCE dieselben sind, und meldet einen Fehler falls die Ausgaben voneinander abweichen. Wie erwähnt haben jedoch alle Superblock-Optimierungen dieses Testframework erfolgreich durchlaufen.

8.1.1. Annotationen

Da der Gesamtumfang der implementierten Optimierungen ca. 18000 Zeilen C++ Quellcode umfasst (Kommentar- und Codezeilen, ohne Leerzeilen), wurden zur Sicherung der Korrektheit des Programmablaufs an diversen Stellen Zusicherungen (Assertions) ins Programm eingestreut. Unter anderen wird nach jedem Optimierungsschritt die Konsistenz der an den Statements annotierten Informationen überprüft:

- Ob `IR_WCETObject`- und `IR_CodeSizeObject`-Annotationen an allen Basisblockköpfen vorhanden sind und an allen anderen Statements *nicht*.
- Ob alle Statements, die in `IR_WCETObject`-, `IR_CodeSizeObject`- und `IR_SuperblockInfo`-Annotationen referenziert werden, Basisblockköpfe sind.
- Ob alle Blöcke eines Traces eine zusammenhängende Kette von Basisblöcken bilden, die jeweils aufeinander verweisen. Dies ist wegen der dezentralen Speicherung der Traces nicht garantiert (s. Abschnitt 6.3).
- Ob alle Pointer innerhalb der Annotationen noch auf gültige Speicherbereiche verweisen.
- Ob alle Annotationen noch mit den ihnen zugewiesenen Statements, die beim Erzeugen der Annotationen noch einmal gesondert in den Annotationen abgespeichert wurden, verknüpft sind. Ohne diese Hilfs-Speicherung existiert zwischen den Annotationen und den Statements nur eine unidirektionale Beziehung, bei der man abfragen kann, welche Annotationen zu einem Statement gehören, aber nicht umgekehrt.

8.1.2. Back-Annotation

Zur Verifikation der Flußerhaltung-Bedingung (Kirchhoff'sche Regel, s. Gleichung 5.8) bezüglich der neu eingeführten Kanten-WCECs (s. Abschnitt 3.4.2) wurde die bestehende BackAnnotation-Testklasse erweitert. An jedem High-Level-Block wird nun unter Beachtung der möglichen Referenzblockbeziehungen überprüft, ob die Summe der eingehenden Kanten-WCECs gleich der der ausgehenden ist. Außerdem wurde ein Test der neu angelegten Annotationskonverter mit in die Testklasse integriert, bei dem alle Annotationen einmal von den Basisblöcken zu den Anweisungen und wieder zurück konvertiert werden, und abschließend überprüft wird, ob die Annotationsdaten unverändert geblieben sind.

8.2. Quantitative Tests

Die Leistungsfähigkeit der neu implementierten Superblock-Optimierungen wurde ebenfalls anhand der bestehenden WCC-Testbench analysiert, wobei Benchmarks ausgeschlossen wurden, die

- in verschiedenen Testsuiten lagen, aber denselben Algorithmus implementierten oder die
- sich nur in der Größe der Eingabedaten voneinander unterschieden haben.

Alle Tests wurden mit der höchsten Optimierungsstufe (O3) durchgeführt, da die Superblockbildung nach Möglichkeit Synergieeffekte bei anderen Optimierungen auslösen soll, und da wir daran interessiert sind, den bereits stark optimierten Code weiter zu verbessern, statt direkt auf dem Eingabecode zu arbeiten. Die Tests wurden für die aktuelle Zielarchitektur des WCC, den Tricore TC1796, kompiliert und für die Analysen und Simulationen wurde der Programmcode im nicht gecachten Flash-Speicher des Tricore abgelegt. Die ACET-Werte wurden mit dem Tricore-Simulator `tsim` der Firma High-Tec ermittelt, wobei die in der WCC Testbench für die Benchmarks verfügbaren Eingabedaten benutzt wurden. Die Position der Superblock-Optimierungen in der Optimierungskette des WCC ist in Abbildung 3.6 dargestellt.

8.2.1. High-Level Superblockbildung

Wir untersuchen zuerst den Nutzen der reinen Superblockbildung, daher werden in diesem Abschnitt jeweils nur die Superblockbildung selbst und die Standard-ICD-C Optimierungen der Optimierungsstufe O3 aufgerufen. Die Superblockbildung ist, so wie sie in dieser Arbeit vorgestellt wurde, in vielen Aspekten konfigurierbar. Mögliche "Stellschrauben" sind dabei:

- Die Codegrößenkontrolle (s. Abschnitt 6.3.2). Standardmäßig sind Grenzen für das Tracewachstum (Faktor 5), das Funktionswachstum (Faktor 3) und das Wachstum der IR (Faktor 2,5) vorgegeben. Diese können jedoch beliebig verändert werden.
- Die Erweiterungen "`else-if`-Neufaltung" (s. Abschnitt 6.4.3) und "`switch`-Konvertierung" (s. Abschnitt 6.4.2) können aktiviert (Standard) oder deaktiviert werden. Bei der "`else-if`-Neufaltung" ist außerdem zu beachten, daß sie standardmäßig nur dann ausgeführt wird, wenn die Heuristiken eine WCET-Verringerung vorhersagen. Da durch zu hohe Schätzwerte die Ausführung der Neufaltung unterdrückt werden kann, wurde zusätzlich eine Möglichkeit eingebaut, die Neufaltung in jedem Fall beim am weitesten innen liegenden `else-if` durchzuführen, was die resultierende Codegröße minimiert.
- Die Anzahl der aiT-Neuberechnungen kann verändert werden (Nutzung bei jeder k-ten WCEP-Neuberechnung, s. Abschnitt 5.3), falls die Neuberechnung mit den Methoden aus Kapitel 5 sich als zu ungenau herausstellt.
- Die Ersetzung von bedingten Ausdrücken im Prepass (s. Abschnitt 6.2) kann aktiviert werden.

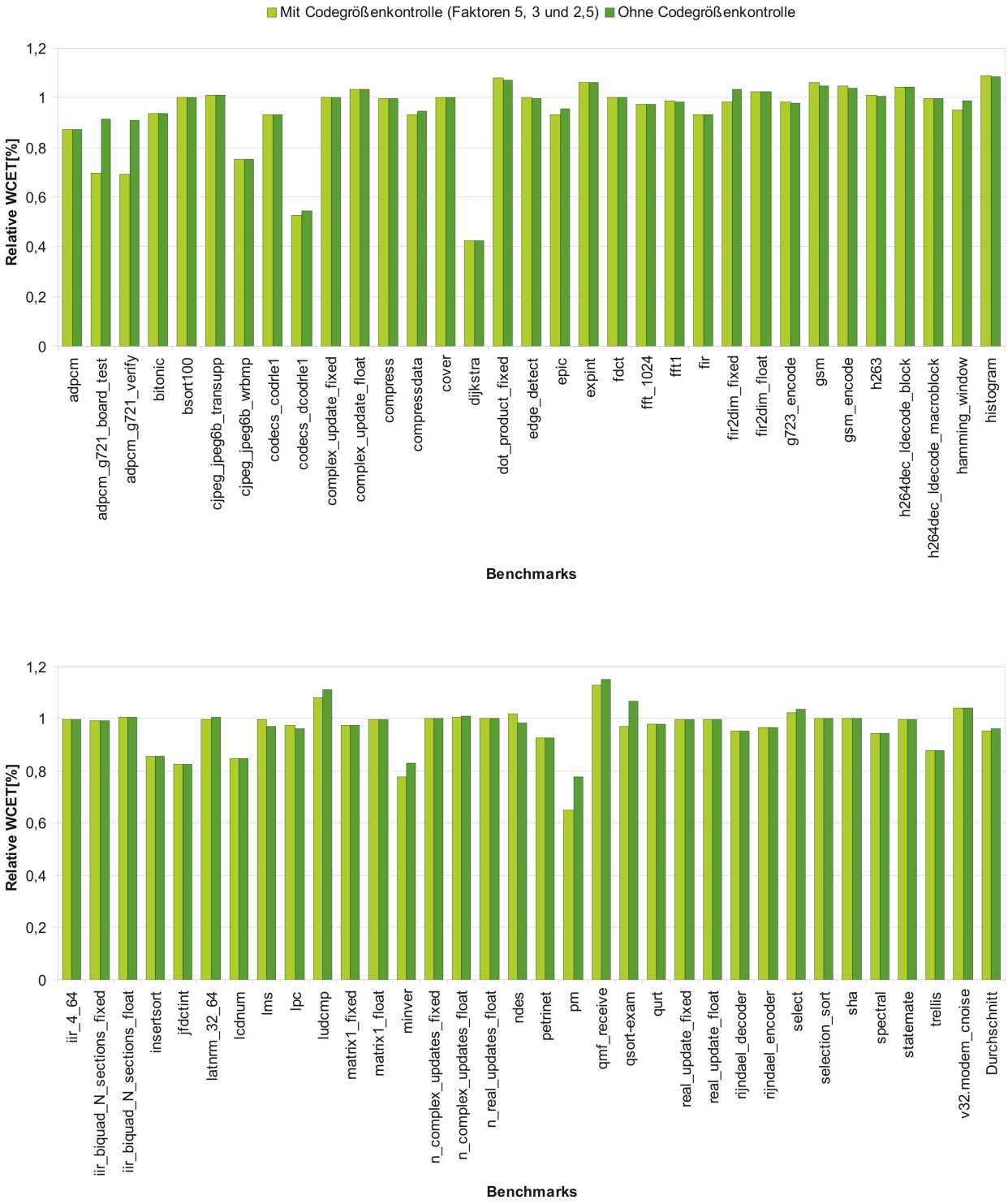


Abbildung 8.1.: WCET-Werte nach der Superblockbildung abh. von der Codegrößenkontrolle (100% $\widehat{=}$ O3)

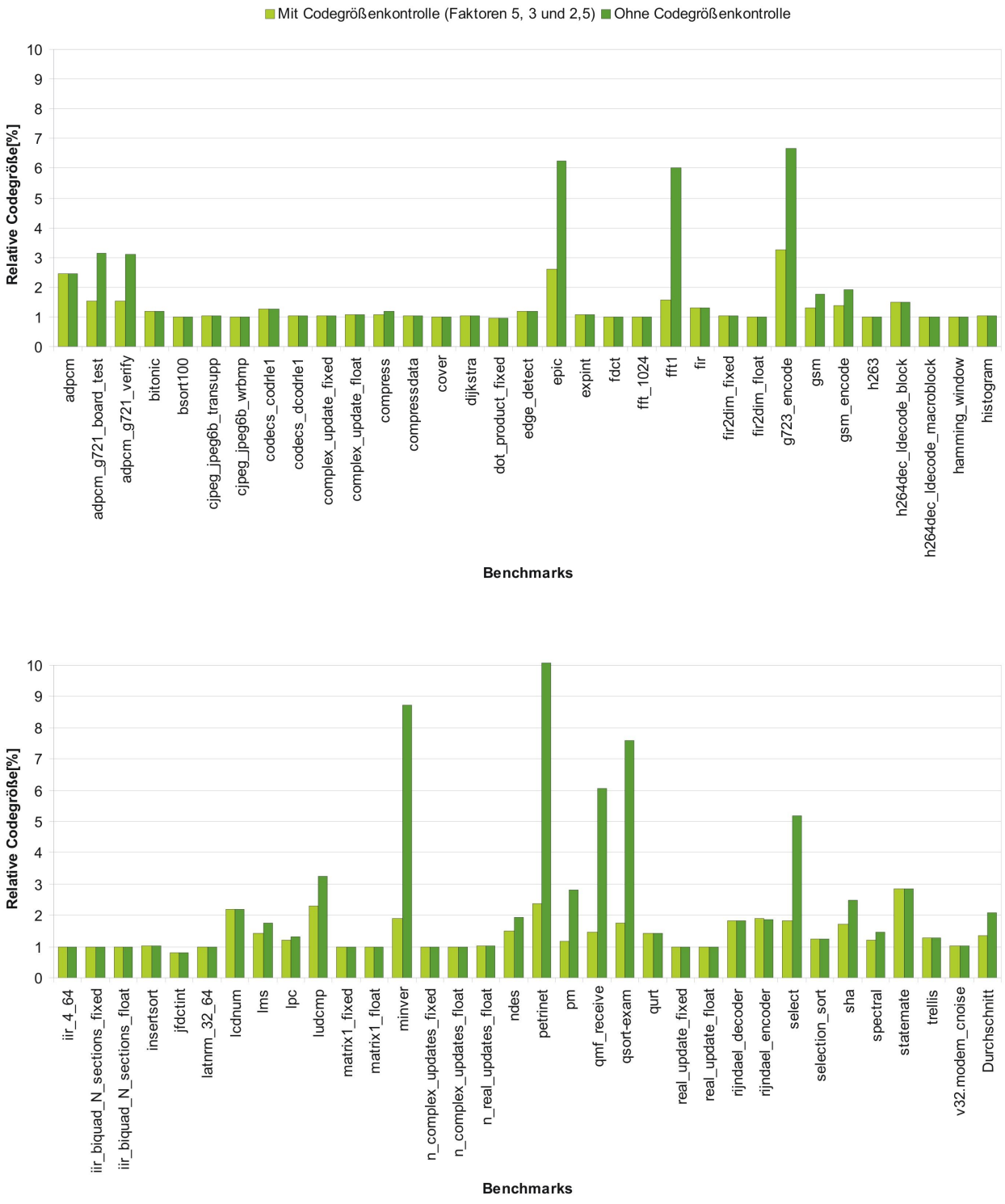


Abbildung 8.2.: Codegrößen nach der Superblockbildung abh. von der Codegrößenkontrolle (100% $\hat{=}$ O3)

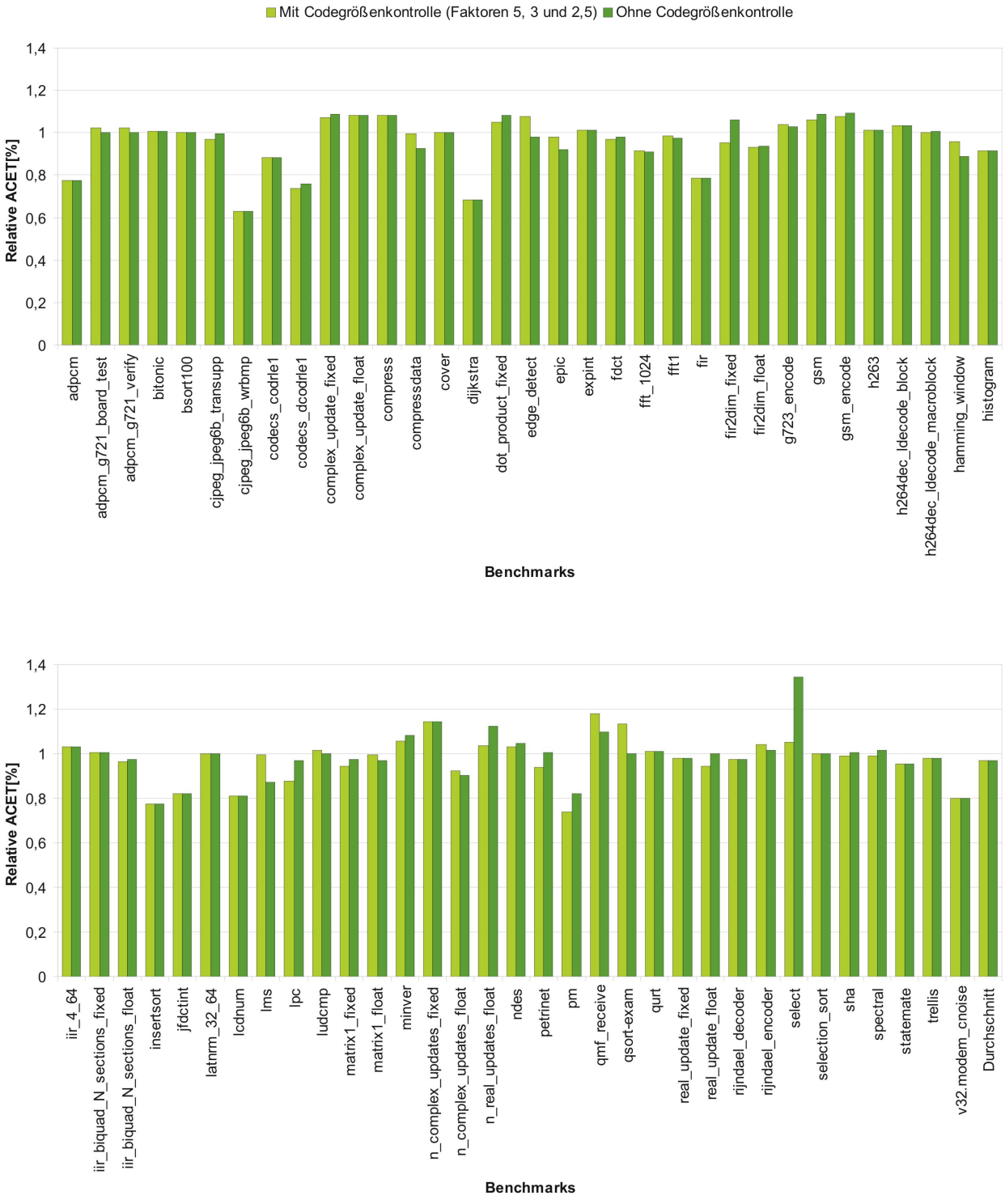


Abbildung 8.3.: ACET-Werte nach der Superblockbildung abh. von der Codegrößenkontrolle (100% $\hat{=}$ O3)

Optimierungsstufe	WCET	ACET	Codegröße	Kompilierzeit
O3 (Referenz)	100%	100%	100%	100%
Superblockbildung mit Codegrößenkontrolle	95,17%	96,76%	134%	640%
Superblockbildung ohne Codegrößenkontrolle	96,38%	97,16%	208%	2650%

Tabelle 8.1.: Ergebnisse der Superblockbildung bei Optimierungslevel O3.

In Tabelle 8.1 sind die WCET-Ergebnisse für die Superblockbildung bei Variation der ersten beiden genannten Optionen aufgelistet. "Mit Codegrößenkontrolle" bedeutet hier, daß die Standardgrenzen (Faktor 5 für Traces, Faktor 3 für Funktionen und Faktor 2,5 für die gesamte IR) verwendet wurden, "Ohne Codegrößenkontrolle" bedeutet, daß die Wachstumsgrenzen auf unendlich gesetzt wurden. Die Ergebnisse bestätigen die Wirksamkeit und Notwendigkeit der Codegrößenkontrolle bei der High-Level-Superblockbildung, da die WCET im Vergleich zum unkontrollierten Fall um 1,21% verbessert wird und 74% des Codegrößenzuwachses sowie ein großer Teil der Kompilierzeit gespart wird wenn die Codegrößenkontrolle aktiv ist (siehe Tabelle 8.1). Die detaillierten WCET-Ergebnisse sind in Abbildung 8.1 dargestellt, die resultierenden Codegrößen sind in Abbildung 8.2 zu finden. Hier ist zu erkennen, daß alle Benchmarks bis auf einen Ausreißer bei `gsm723_encode` den Faktor 2,5 bei der Codevergrößerung erfolgreich einhalten. Der Ausreißer ist höchstwahrscheinlich damit zu erklären, daß die High-Level Superblockbildung die Größe des resultierenden Codes mithife der Back-Annotation-Daten nur ungefähr abschätzen kann. Low-Level-Optimierungen und Transformationen die erst nach der Superblockbildung durchgeführt werden, können diese Schätzung stören. Die ACET-Ergebnisse sind im Detail in Abbildung 8.3 zu finden und bestätigen die Ergebnisse der WCET-Analysen insofern, als daß sich die ACETs nach der Superblockbildung mehr oder weniger genauso entwickeln wie die WCETs. Die stark vergrößerte Kompilierzeit ist im wesentlichen auf die wiederholten Analysen mit aiT zurückzuführen. Da wir auf der High-Level-Ebene arbeiten, müssen vor jeder aiT-Analyse auch die Codeselektion, die LLIR-Optimierungen und die Registerallokation sowie das Instruction Scheduling neu durchgeführt werden, was die Kosten einer Neuberechnung mit aiT weiter in die Höhe treibt. Eine genauere Darstellung des Zusammenhangs zwischen Kompilierzeit und aiT-Nutzung ist in den weiter unten präsentierten Ergebnissen bezüglich verschiedener aiT-Nutzungsintervalle enthalten.

Es ist davon auszugehen, daß einzelne, starke Verbesserungen der WCET, wie z.B. bei `dijkstra`, darauf zurückzuführen sind, daß aiTs Analysen nach der Superblockbildung in der Lage sind, genauere Informationen zu ermitteln. Im Beispiel von `dijkstra` wurden in der Funktion, die den größten Teil der WCET verursacht, alle Blöcke mit mehr als einer eingehenden Kontrollflußkante in Blöcke mit nur einer solchen Kante transformiert. Wie wir schon in Algorithmus 7.2 gesehen haben, müssen bei solchen Verschmelzungspunkten auch die Datenflußinformationen miteinander verschmolzen werden, wobei eine Überapproximation (das Supremum der beteiligten Informationen) gebildet werden muß. Durch die Superblockbildung wurden im Falle dieser speziellen Funktion alle Verschmelzungspunkte eliminiert, und dies führt dazu, daß für die einzige Schleife der erwähnten Funktion eine wesentlich präzisere WCET-Abschätzung bestimmt werden kann, obwohl der Assemblercode der Schleife sich nicht verändert hat. Um diesen Effekt näher zu untersuchen müsste man die Details der aiT-Analysen einsehen können. Da dies jedoch für uns nicht möglich ist, kann nicht abschließend geklärt werden, warum genau aiT zu dieser besseren Abschätzung kommt.

Für die teilweise auftretenden WCET-Steigerungen in Abbildung 8.1, wie z.B. bei `qmf_receive`, gibt es im Wesentlichen zwei Erklärungsansätze:

- Wir haben durch falsche Schätzungen bezüglich der Basisblock-WCETs den eigentlichen WCEP verlassen und optimieren auf einem anderen Pfad zu Lasten des WCEP.

Optimierungsstufe	WCET	ACET	Codegröße	Kompilierzeit
O3 (Referenz)	100%	100%	100%	100%
Superblockbildung (k=2, hohe aiT-Nutzung)	95,34%	96,92%	136%	857%
Superblockbildung (k=4, normale aiT-Nutzung)	95,17%	96,76%	134%	640%
Superblockbildung (k=8, geringe aiT-Nutzung)	95,81%	96,91%	132%	472%

Tabelle 8.2.: Ergebnisse der Superblockbildung bei verschiedener aiT-Nutzung.

- Nach der Superblockbildung treten weitere Effekte auf, die z.B. durch nachfolgende High- oder Low-Level-Optimierungen oder das Speicher-Layout hervorgerufen werden können. Es ist bekannt, daß Optimierungen miteinander auf komplexe Weise interagieren. Diese Interaktionen sind aber kaum vorhersagbar, so daß oft auch negative Effekte von Optimierungen nicht vermeidbar sind (s. [Davidson & Hollersnm, 1992] und [Cooper et al., 1991] für eine Analyse dieses Problems bei der High-Level-Optimierung "Function Inlining"). Denkbar ist z.B. daß nach der Superblockbildung durch das geänderte Speicher-Layout andere Sprunginstruktionen benötigt werden, die die WCET in die Höhe treiben, oder daß zusätzliche Line-Crossing-Effekte beim Zugriff auf die verschobenen Instruktionen auftreten.

Am zweiten Punkt lässt sich aus Sicht der High-Level-Optimierungen wenig verändern, die Vorraussage der Effekte, die die Low-Level-Optimierungen haben werden, ist faktisch nicht machbar. Um den ersten Punkt zu untersuchen, wurde das aiT-Neuberechnungsintervall variiert, so daß aiT in jeder 2., 4. oder 8. Neuberechnung des WCEP benutzt wurde. Falls der erste Punkt zutreffend sein sollte, müsste sich die erzielte WCET bei einer erhöhten Anzahl von aiT-Nutzungen verbessern. In Tabelle 8.2 und Abbildung 8.4 sind die Ergebnisse dieser Tests aufgeführt, aus denen relativ deutlich hervorgeht, daß eine Fehlberechnung des WCEP nicht der Grund für die WCET-Steigerungen sein kann, da die WCET-Werte sich bei hoher aiT-Nutzung nicht verbessern. Für die Zukunft wäre es im Gegenteil eher ratsam die Anzahl der Neuberechnungen durch aiT zu reduzieren, da sich die WCET selbst bei Nutzungsintervall $k = 8$ kaum verschlechtert, die Kompilierzeit jedoch stark verbessert wird. In allen folgenden Tests, bei denen das Intervall k nicht mehr explizit angegeben wird, wurde die Standardeinstellung $k = 4$ verwendet.
Im normalen Ablauf der Superblockoptimierungen werden die WCEP-Neuberechnungen erst aufgerufen, wenn bereits ein vollständiger Superblock gebildet wurde (s. Abbildung 6.1). Als Erweiterung hierzu wurde für die Tests die Möglichkeit geschaffen, den WCEP nach der Elimination jedes *einzelnen* Einsprungpunktes im Superblock neu zu berechnen, und die Superblockbildung unter Beachtung des evtl. veränderten WCEPs fortzusetzen. Falls also Pfadwechsel, die *während* der Superblockbildung auftreten, der Grund für die WCET-Steigerungen sein sollten, so müsste sich bei der Neuberechnung nach jeder Einsprungpunkt-Elimination eine WCET-Verbesserung gegenüber der Standard-Superblockbildung ergeben. Dies war jedoch, analog zu den Ergebnissen aus Tabelle 8.2 *nicht* der Fall, so daß es sehr unwahrscheinlich scheint, daß die schlechte Performance auf einigen Benchmarks an verborgenen Pfadwechseln liegt. Die teilweise auftretende Erhöhung der WCET geht auch meist mit einer Erhöhung der ACET um ca. denselben Faktor einher (s. Abbildung 8.3). In dieser Hinsicht decken sich unsere Ergebnisse mit denen von Kidd [Kidd & Hwu, 2006] (Medium-Level-Superblockbildung und -Scheduling), da auch dort einzelne Benchmarks durch die Optimierung negativ beeinflußt wurden, auch wenn der Durchschnitt der Benchmarks davon profitierte.

Im Zusammenhang mit der Superblockbildung wurde außerdem untersucht, wie sich die implemen-

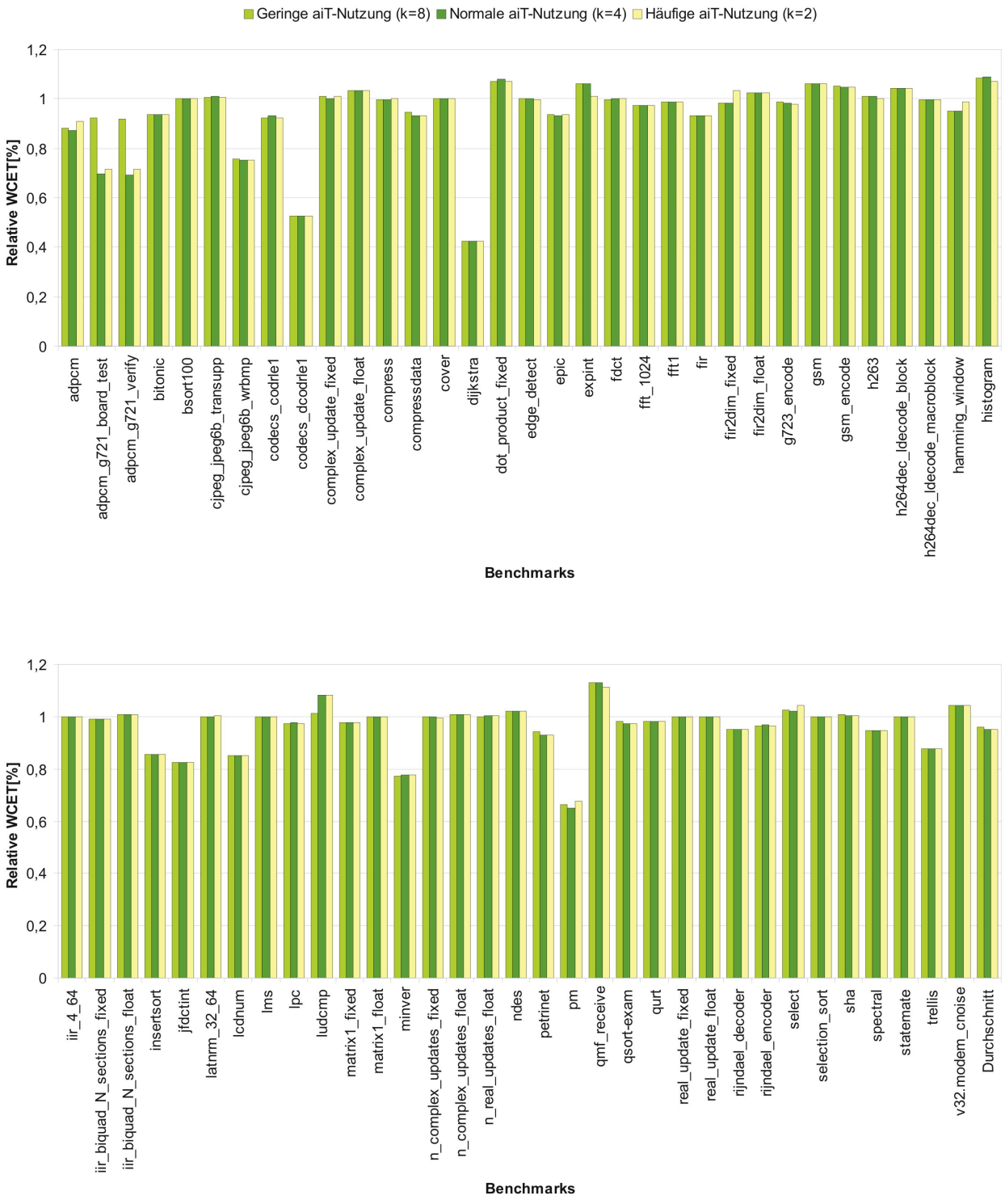


Abbildung 8.4.: WCET-Werte nach der Superblockbildung abh. von der aiT-Nutzungshäufigkeit (100% $\widehat{=}$ O3)

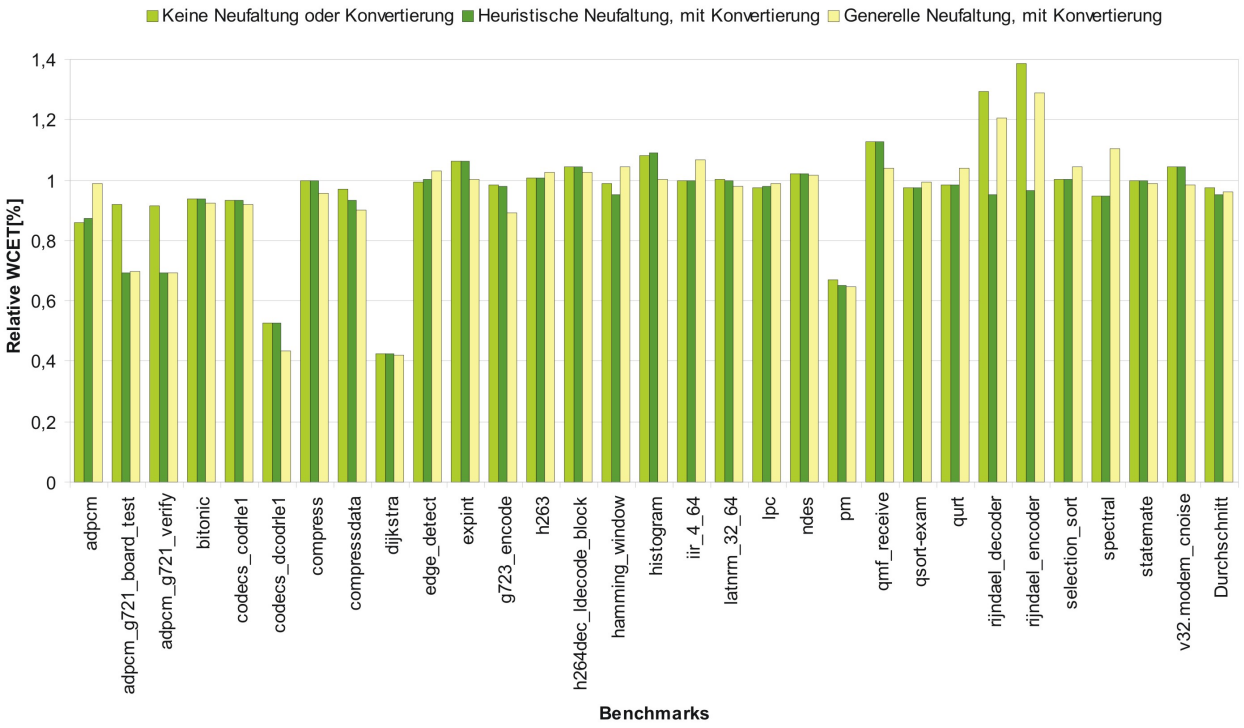


Abbildung 8.5.: WCET-Werte nach der Superblockbildung mit `else-if`-Neufaltung und `switch`-Konvertierung (100% ≙ O3)

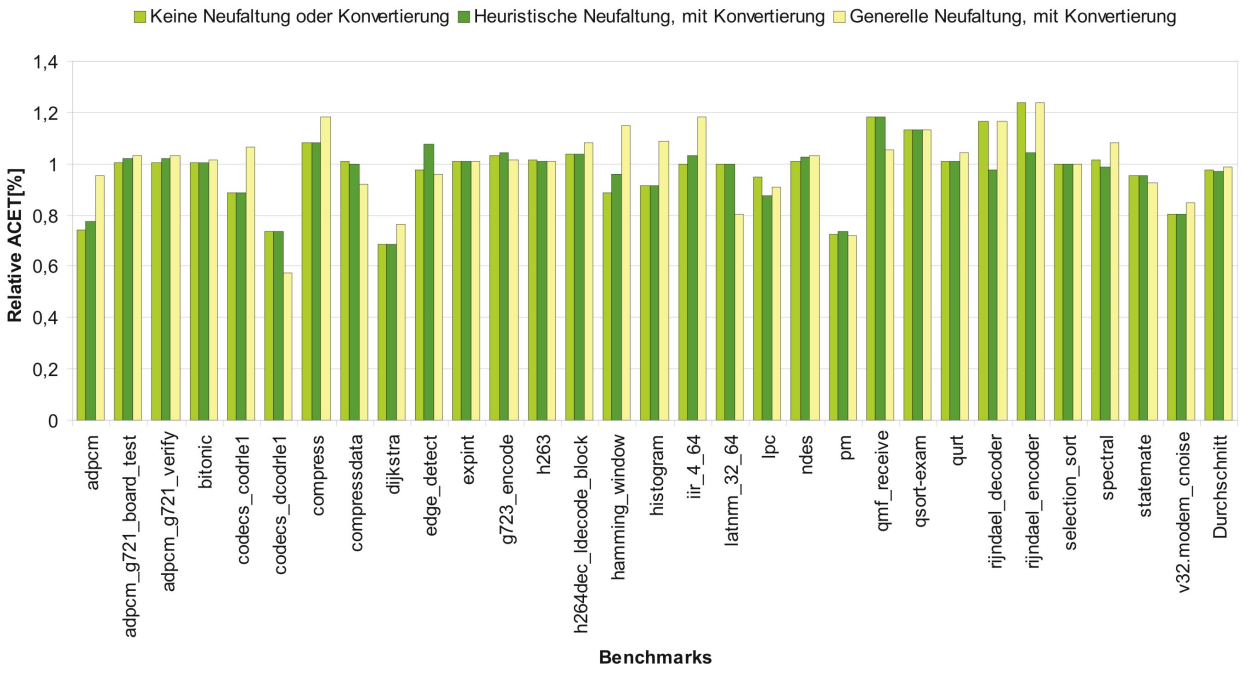


Abbildung 8.6.: ACET-Werte nach der Superblockbildung mit `else-if`-Neufaltung und `switch`-Konvertierung (100% ≙ O3)

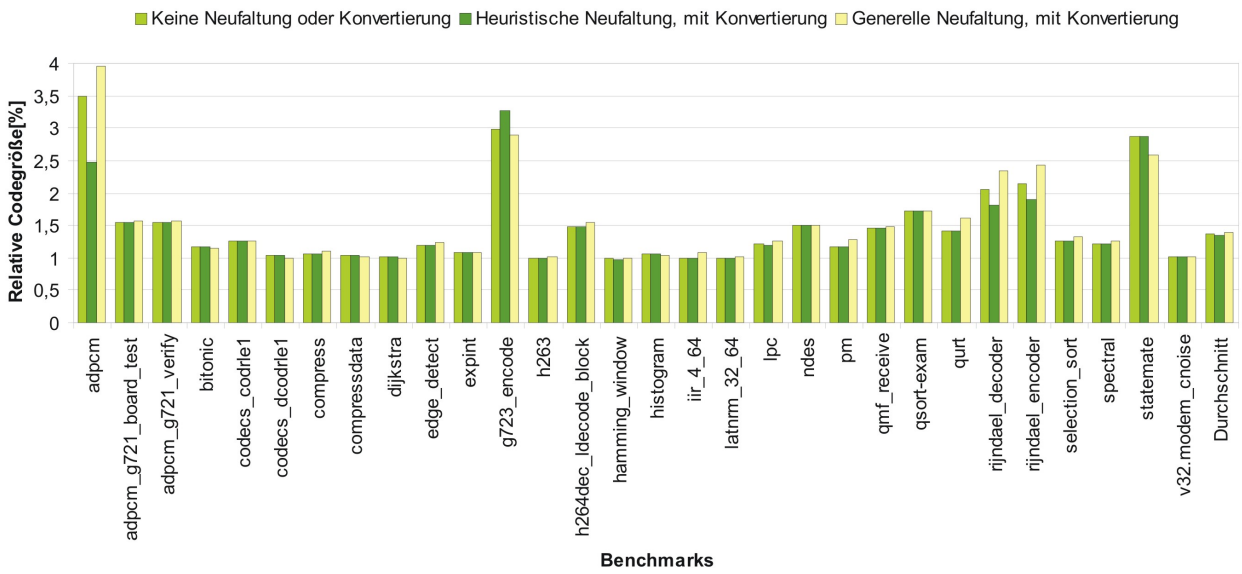


Abbildung 8.7.: Codegrößen nach der Superblockbildung mit `else-if`-Neufaltung und `switch`-Konvertierung (100% $\widehat{=}$ O3)

Optimierungsstufe	WCET	ACET	Codegröße
O3 (Referenz)	100%	100%	100%
Superblockbildung ohne Erweiterungen	97,29%	97,59%	137%
Superblockbildung + SK + HNF	95,17%	96,76%	134%
Superblockbildung + SK + GNF	96,17%	98,65%	139%

Tabelle 8.3.: Ergebnisse der `else-if`-Neufaltung und `switch`-Konvertierung.

tierten Erweiterungen (`else-if`-Neufaltung und `switch`-Konvertierung) auf die Ergebnisse auswirken. Dazu wurde ein erster Lauf der Superblockbildung mit deaktivierter `else-if`-Neufaltung und `switch`-Konvertierung durchgeführt. In einem nächsten Lauf wurden beide Erweiterungen aktiviert, die Neufaltung allerdings nur in der Version, wie sie in Abschnitt 6.4.3 vorgestellt wurde. In einem dritten Lauf wurden dann beide Erweiterungen aktiviert und die Neufaltung wurde nicht durch die Heuristik zur Verringerung der WCET aus Abschnitt 6.4.3 gesteuert, sondern wurde immer ausgeführt wenn dies prinzipiell möglich war. Die WCET-, ACET- und Codegrößen-Ergebnisse hierzu sind in den Abbildungen 8.5, 8.6 und 8.7 zu finden. In Tabelle 8.3 sind die durchschnittlichen Ergebnisse noch einmal zusammengefaßt. Die Abbildungen zeigen nur die Untermenge von Benchmarks für die die beiden Erweiterungen überhaupt einen Effekt hatten, die Tabelle hingegen zeigt den Durchschnitt über alle Benchmarks. "SK" steht hier für "`switch`-Konvertierung", "HNF" für "Heuristische Neufaltung" und "GNF" für "Generelle Neufaltung", wie oben erwähnt. Es zeigt sich, daß die eingesetzte Heuristik einen guten Mittelweg für die Anwendung oder Nichtanwendung der Neufaltung findet, da beide anderen Extremfälle schlechtere WCET-Werte liefern.

Weiterhin wurde versucht, durch ein aggressiveres Unrolling und Inlining mehr Code zu schaffen auf dem die Superblockbildung arbeiten kann. Dazu wurden die Grenzen für die Anzahl der Ausdrücke, die in einem Schleifenrumpf oder einer Funktion vorkommen dürfen, damit die Schleife oder Funktion noch für das Unrolling / Inlining in Frage kommt, von 50 bzw. 20 Ausdrücke auf je 150 Ausdrücke erhöht und anschließend eine Superblockbildung durchgeführt. Ähnlich wie beim Verzicht auf die Codegrößenkontrolle führte dies jedoch nicht zu einer Verbesserung, sondern im

Optimierungsstufe	WCET	ACET	Codegröße
O3 (I-150,U-150) (Referenz)	100%	100%	100%
Superblockbildung (I-150,U-150)	98,49%	99,02%	134%

Tabelle 8.4.: Ergebnisse der Superblockbildung bei Optimierungslevel O3 mit aggressivem Inlining/Unrolling.

Optimierungsstufe	WCET	ACET	Codegröße	Kompilierzeit
O3 ohne ICD-C CSE (Referenz)	100%	100%	100%	100%
O3 mit ICD-C CSE	102,22%	98,63%	93%	106%
O3 ohne ICD-C CSE mit SB-Bildung	98,70%	97,91%	135%	683%
O3 ohne ICD-C CSE mit SB-Bildung und SB-CSE	95,28%	92,95%	125%	1285%

Tabelle 8.5.: Ergebnisse der Superblock-CSE.

Durchschnitt zu einer Verschlechterung der WCET. Die Ergebnisse hierzu sind in Tabelle 8.4 aufgeführt. Solche negativen Effekte bei zu aggressivem Unrolling und Inlining sind in der Literatur bereits bekannt ([Davidson & Hollersnm, 1992], [Cooper et al., 1991]).
Die Ersetzung *aller* bedingten Ausdrücke im Prepass (s. Abschnitt 6.2) hat zu ähnlichen Ergebnissen geführt, da hierbei zu viele neue `if`-Konstrukte geschaffen werden und der Code somit unnötig aufgebläht wird. Im Vergleich dazu hat die Ersetzung aller derjenigen bedingten Ausdrücke, die Schreibzugriffe enthalten, kaum einen Unterschied zur normalen Superblockbildung ergeben, was allerdings daran liegt, daß in der aktuellen Testbench solche Ausdrücke kaum vorkommen (Spalte L in Anhang A).
In allen folgenden Tests werden daher wieder die Standard-Parameter für das Inlining / Unrolling benutzt, die auch in den Tests aus Tabelle 8.1 benutzt wurden, und die Ersetzung bedingter Ausdrücke im Prepass wird deaktiviert.

8.2.2. High-Level Superblock-CSE

Bei der High-Level Superblock-CSE haben wir keine Optionen implementiert, so daß hier im Wesentlichen nur ein einzelner Testlauf nötig ist. Wir vergleichen hierbei die Performance der Superblock-CSE mit der der ICD-C CSE, die nur auf Basisblockebene arbeitet. Der Referenzwert ist die WCET des Benchmarks bei Kompilierung mit Optimierungsstufe O3, *ohne* die ICD-C CSE. Ausgehend davon ist die Veränderung der WCET bei Aktivierung der ICD-C CSE (1. Säule), der Superblockbildung (2. Säule) und der Superblockbildung und Superblock-CSE (3. Säule) in Abbildung 8.8 dargestellt. Die WCET-Durchschnittswerte sind in Tabelle 8.5 abgebildet. Die Superblock-CSE ist dabei in der Lage, die WCET gegenüber der ICD-C Standard-CSE um durchschnittlich 6,94%, und die ACET um 5,68% zu verbessern, bei einem zusätzlichen Codegrößenwachstum um 32%. Eine detaillierte Übersicht über die ACET-Veränderungen ist in Abbildung 8.9 zu finden. Hier lässt sich erkennen, daß die stark gestiegenen WCET-Werte bei `epic` oder `gsm` höchstwahrscheinlich auf eine Überschätzung der WCET durch aiT zurück gehen, da sich die ACET-Werte nach der CSE bei beiden Benchmarks positiv entwickeln. Die stark erhöhte Kompilierzeit wird vor allem durch die größeren Benchmarks verursacht, der Median der Kompilierzeiten liegt beispielsweise bei 344%. Diese Zeiten rühren wahrscheinlich zumindest zum Teil daher, daß die Verwaltung der verfügbaren Ausdrücke und die Berechnung der Def/Use-Sets mit Hilfe von C++ STL-Sets erfolgt. Bei der Verwendung von Bitvektoren ließen sich die dort oft ausgeführten Mengenschnitt- und Mengenvereinigungs-Operationen wesentlich effizienter realisieren.

Die teilweise auftretende Erhöhung der ACET, wie beispielsweise bei den `ndes`-, `selection_sort`- und `compress`-Benchmarks, wird meist durch eine zu hohe Anzahl von erzeugten temporären Variablen für die Zwischenspeicherung bewirkt. Diese temporären Variablen werden durch den Codeselektor zu zusätzlichen virtuellen Registern in der LLIR. Hierdurch kommt es gerade bei größeren Benchmarks zu vermehrter Spillcode-Generierung bei der Registerallokation, da die Anzahl der gleichzeitig lebendigen Register erhöht wird. Daher hat, wie im Diagramm ersichtlich, auch die ICD-C CSE in allen diesen Fällen mit demselben Problem zu kämpfen. Der einzige Ausweg wäre die Back-Annotation der Anzahl der lebendigen Register pro Basisblock, auf Grund derer die CSE entscheiden könnte, ob die Generierung neuer Variablen zu Spillcode führen würde. Eine solche Vorhersage würde aber vorraussetzen, daß vor jedem CSE-Durchlauf eine erneute Codeselektion, LLIR-Optimierung und Registerallokation durchgeführt würde. Da diese Komponenten bei größeren Benchmarks eine nicht zu vernachlässigende Laufzeit haben, wurde dieses Verfahren bisher nicht umgesetzt.

Die Gründe für das teilweise schlechte Abschneiden bezüglich der WCET sollen hier noch einmal am konkreten Beispiel des `adpcm_g721_board_test`-Benchmarks besprochen werden. Dort treffen mehrere Faktoren aufeinander, die zur negativen Wirkung der CSE bei diesem Benchmark beitragen:

- **Die Kosten-Nutzen Relation**. Die Zwischenspeicherung ist teilweise teurer als die Neuberechnung. Dies trifft dann zu, wenn das zwischengespeicherte Ergebnis nicht sehr oft wiederverwendet wird und die ersparte Berechnung gleichzeitig sehr simpel war. Im Benchmark `adpcm_g721_board_test` gibt es mehrere solcher Beispiele, eines davon ist das folgende:

  ```
  DL=(EXP«7)+(((-7)+EXP<0 ? DQM«-((-7)+EXP) : DQM»(-7)+EXP)&127);
  ```

  wird dort zu

  ```
  DL=(EXP«7)+(((temp=(-7)+EXP, temp)<0 ? DQM«-temp : DQM»temp)&127);
  ```

  `temp` wird hierbei sonst nie wieder verwendet. Die WCET des Basisblock steigt dabei über die berechneten 32 Ausführungen (WCEC) um insgesamt 416 Zyklen an.

- **Die Registerallokation**. Dies wurde zwar oben bereits erwähnt, soll aber hier noch einmal an einem konkreten Beispiel erläutert werden. In `adpcm_g721_board_test` wird in der Funktion `adpt_predict_1` eine weitere Ersetzung durchgeführt, so daß dort eine neue Variable für die Zwischenspeicherung eingefügt werden muß. Dadurch steigt die WCET eines Blocks in einer inneren Schleife, der von der CSE selbst gar nicht direkt verändert wird, um insgesamt 104244 Zyklen (die WCET des gesamten Benchmarks betrug vorher 295392 Zyklen), so daß sämtliche durch die CSE erarbeiteten positiven Effekte überdeckt werden. Für die CSE ist dies nicht vorhersehbar.

- **Überschätzungen durch aiT**. Angesichts des enormen Zuwachses der WCET bei Benchmarks wie `adpcm_g721_board_test`, wobei die ACET durch die Optimierung jedoch *sinkt*, ist es enorm unwahrscheinlich, daß dieser WCET-Zuwachs wirklich komplett durch die Registerallokation ausgelöst wird. Vielmehr ist davon auszugehen, daß aiT hier durch eine kleine Änderung am Programm, z.B. an der konkreten Registerallokation, zu stark vergrößerten Schätzwerten kommt. Warum dies der Fall ist lässt sich schon von der Low-Level-Ebene aus kaum analysieren, da aiT nur als Black-Box genutzt wird. Auf der High-Level-Ebene ist es völlig unmöglich solche Effekte vorrauszusagen. Eine solche Überschätzung scheint z.B. auch beim `epic`-Benchmark vorzuliegen, da dort die WCET der Funktion `internal_filter` fast verdoppelt wird, obwohl nur eine einzige neue Variable eingefügt wird. Da auch hier die

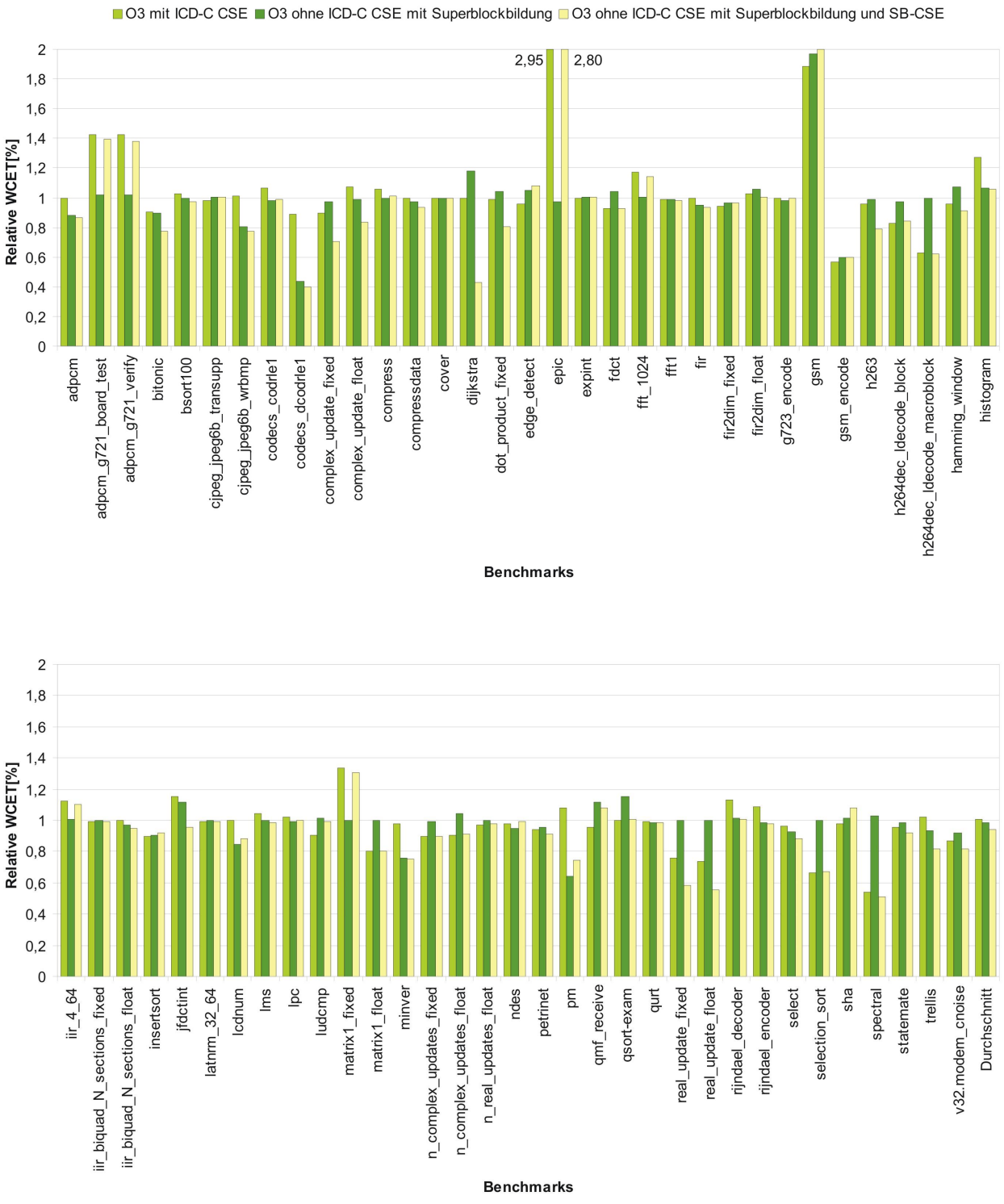


Abbildung 8.8.: WCET-Werte der verschiedenen CSE-Algorithmen (100% ≙ O3 *ohne* ICD-C CSE)

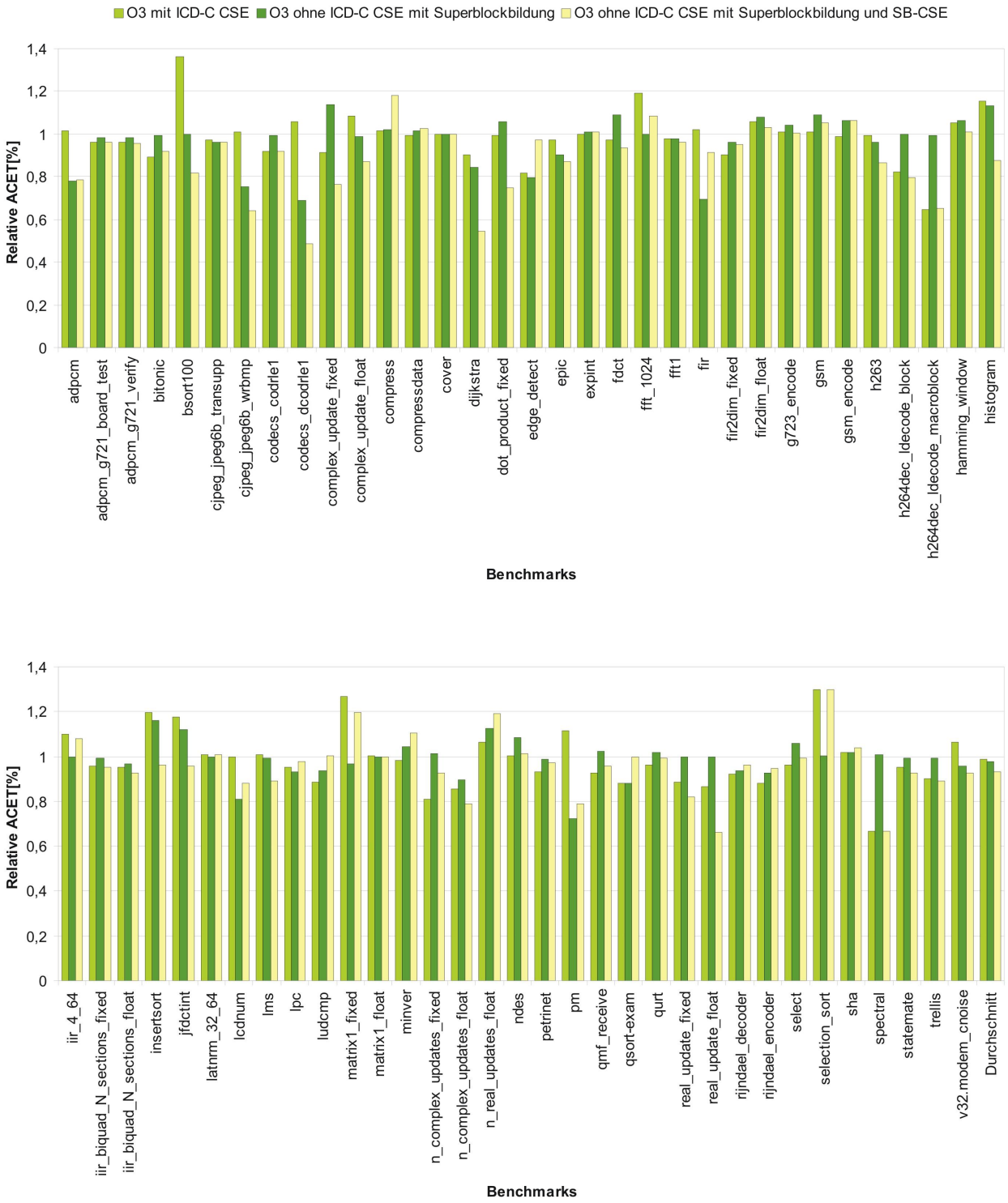


Abbildung 8.9.: ACET-Werte der verschiedenen CSE-Algorithmen (100% $\hat{=}$ O3 *ohne* ICD-C CSE)

Optimierungsstufe	WCET	ACET	Codegröße	Kompilierzeit
O3 ohne ICD-C DCE (Referenz)	100%	100%	100%	100%
O3 mit ICD-C DCE	99,97%	101,06%	99%	105%
O3 ohne ICD-C DCE mit SB-Bildung	97,93%	97,40%	136%	707%
O3 ohne ICD-C DCE mit SB-Bildung und SB-DCE	95,32%	94,36%	131%	1162%

Tabelle 8.6.: Ergebnisse der Superblock-DCE.

ACET des Benchmarks durch die CSE gesenkt wird, ist nicht davon auszugehen, daß diese große WCET-Steigerung durch den Registerallokator ausgelöst wird.

Da die Superblock-CSE die Ersetzungen nur entlang des WCEP durchführt, ist sie von diesen Effekten in einigen Fällen weniger stark betroffen als die allgemeine CSE, wie man am Beispiel des `jfdctint` oder `trellis` Benchmarks sehen kann (s. Abbildung 8.8).

8.2.3. High-Level Superblock-DCE

Bei der Bewertung der Performance der Superblock-DCE gehen wir ähnlich wie bei der Superblock-CSE vor. Wir führen dazu einen Testlauf durch, in dem die Werte für Optimierungslevel O3 *ohne* ICD-C DCE gemessen werden (Referenzwert). Die Ergebnisse der Kompilierung auf O3 *mit* ICD-C DCE (1. Säule), auf O3 ohne ICD-C DCE aber mit Superblockbildung (2. Säule) und auf O3 ohne ICD-C DCE aber mit Superblockbildung und Superblock-DCE (3. Säule) werden im Vergleich zum Referenzwert gemessen. Die WCET-Ergebnisse pro Benchmark sind in Abbildung 8.10 zu sehen, die durchschnittlichen WCET-Ergebnisse in Tabelle 8.6. Die Superblock-DCE erreicht dabei eine durchschnittliche Verbesserung um 4,65% gegenüber der Standard-DCE. Eine detaillierte Auflistung der ACET-Ergebnisse ist in Abbildung 8.11 zu finden. Hier scheint die WCET-Vergrößerung bei `h264dec_ldecode_block` gerechtfertigt zu sein, und nicht aus einer Überschätzung durch aiT zu resultieren. Die Gründe für die erhöhte Kompilierzeit sind hier ähnlich wie bei der CSE, allerdings nimmt auch hier die Kompilierzeit der größeren Benchmarks überproportional zu - der Median der Kompilierzeiten mit SB-DCE liegt z.B. bei nur 312% statt 1162%. Bei der DCE wirkt insbesondere die Implementierung der Lebensdauer-Analyse auf Grundlage von STL-Sets negativ, da diese Analyse von der DCE jedesmal aufgerufen werden muß und ihrerseits eine große Anzahl von Mengenvereinigungen und -schnitten vornehmen muß (s. Algorithmus 7.2). Daher würde die DCE wohl am meisten von einer Umsetzung der Symbolmengen als Bitvektoren profitieren.

Die bei der ICD-C DCE teilweise auftretenen Steigerungen der WCET haben vor allem den Grund, daß die ICD-C DCE `do-while`-Schleifen der Form

```
do {
  ...
} while (j=j+1, j<=5);
```

zu

```
do {
  ...
  j=1+j;
} while (j<=5);
```

transformiert, was im WCC z.B. beim `ludcmp`-Benchmark bezüglich der WCET zu bis zu 30% schlechterem Assemblercode führt. Die Verschlechterung ist auch an den ACET-Werten (s. Abbildung 8.11) zu erkennen. Die teilweise ebenfalls vorkommenden Steigerungen der WCET bei

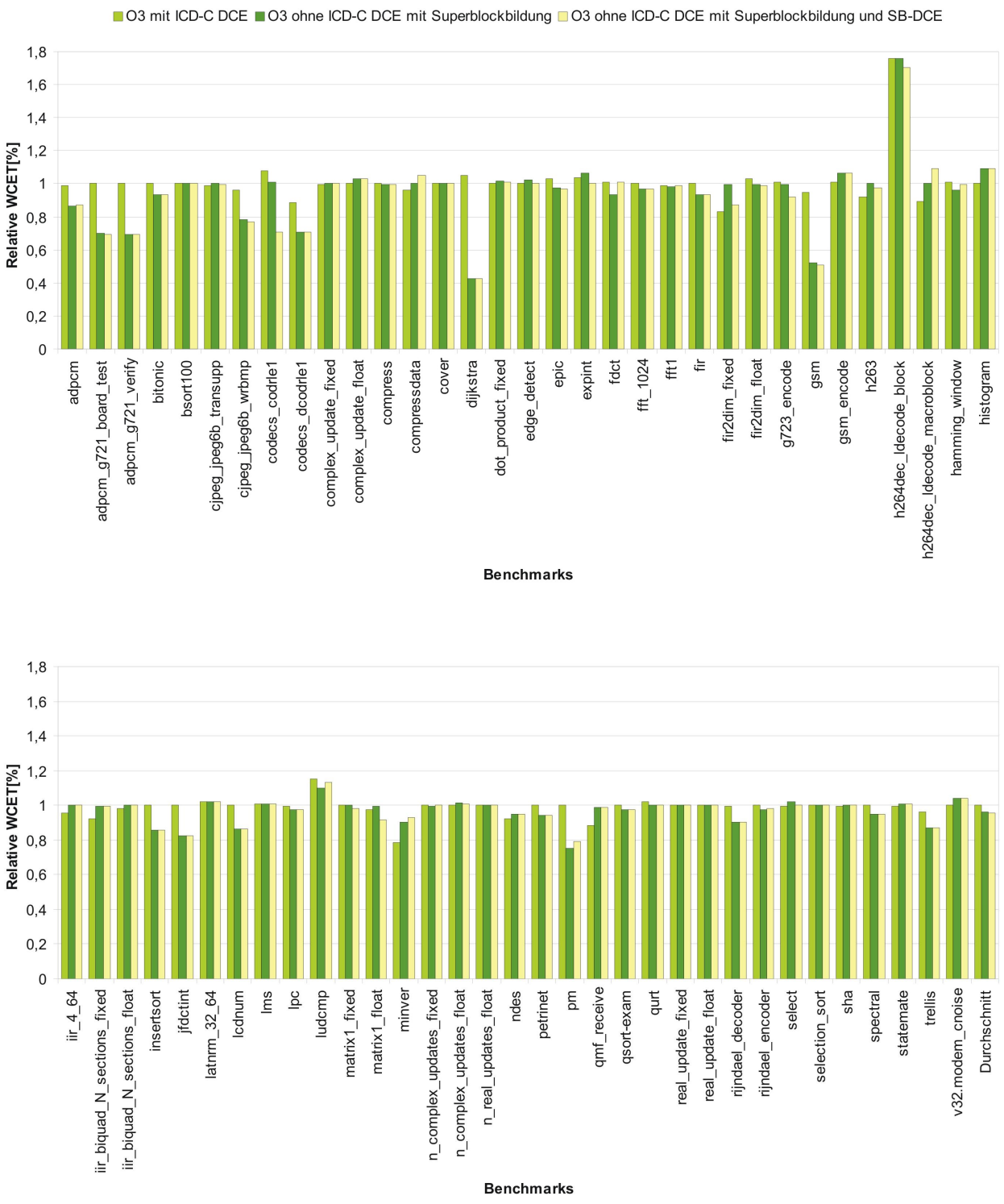


Abbildung 8.10.: WCET-Werte der verschiedenen DCE-Algorithmen (100% $\widehat{=}$ O3 *ohne* ICD-C DCE)

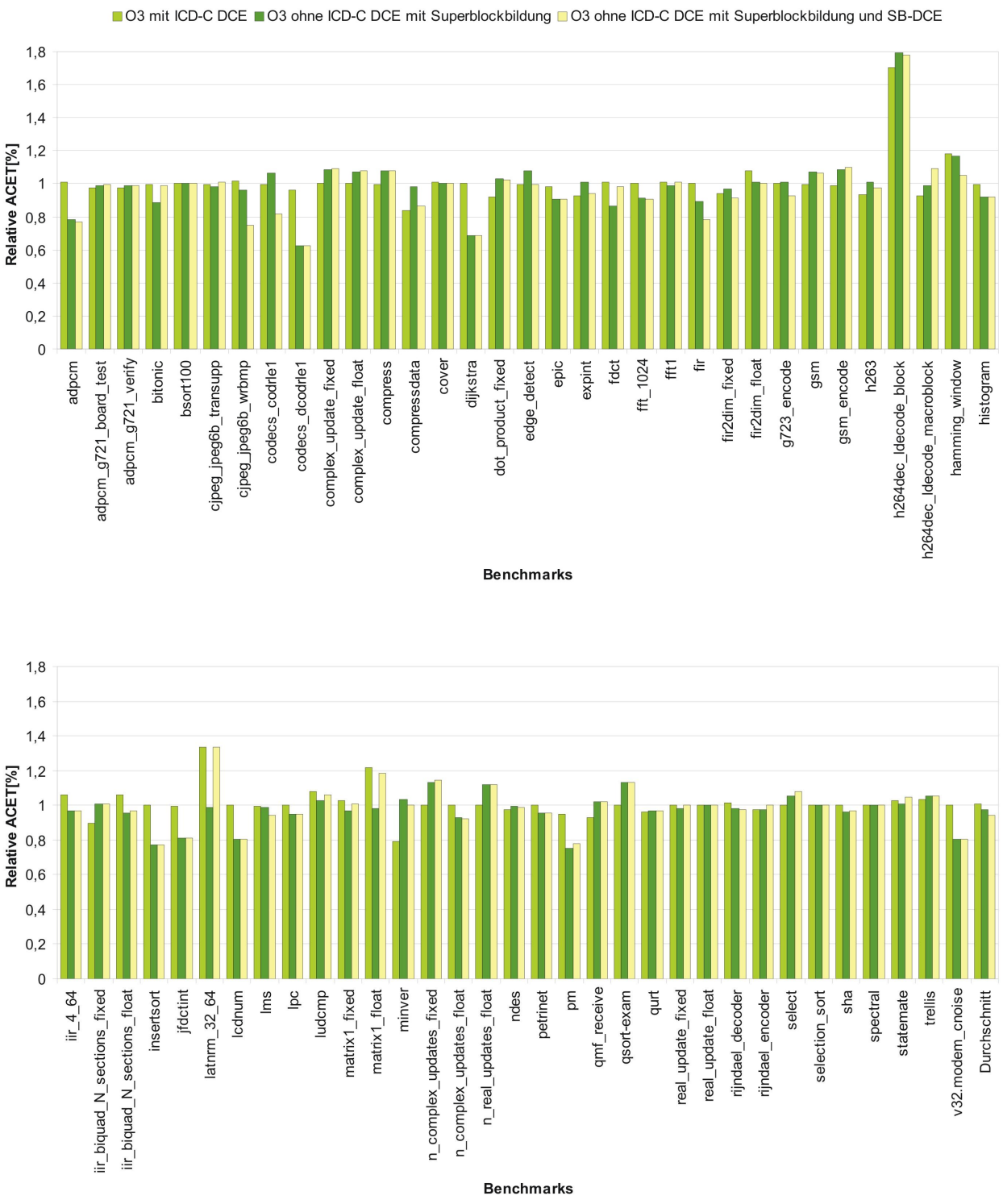


Abbildung 8.11.: ACET-Werte der verschiedenen DCE-Algorithmen (100% $\widehat{=}$ O3 *ohne* ICD-C DCE)

Optimierungsstufe	WCET	ACET	Codegröße
O3 ohne ICD-C DCE (Referenz)	100%	100%	100%
Superblock-Bildung und SB-DCE	95,32%	94,36%	131%
Superblock-Bildung, SB-DCE und Code Sinking	97,30%	98,11%	145%

Tabelle 8.7.: Ergebnisse des Code Sinkings.

der Superblock-DCE sind z.B. im Benchmark `fdct` kaum zu erklären. Dort wird eine Folge von Konstanten, die durch die Value Propagation unnötig geworden sind, eliminiert. Dabei wird aus

```
constant=4433;
z1=((__80+__81)*4433);
constant=6270;
block[__65]=(z1+(6270*__81))>>18;
constant=(-15137);
block[__67]=(z1+((-15137)*__80))>>18;
```

die Anweisungsfolge

```
z1=((__379+__380)*4433);
block[__364]=(z1+(6270*__380))>>18;
block[__366]=(z1+((-15137)*__379))>>18;
```

Trotzdem steigt die WCET des betroffenen Basisblocks in der aiT-Analyse um 5%, was für die High-Level Optimierung kaum nachvollziehbar ist. Die ICD-C DCE löscht zwar bei diesem Benchmark exakt dieselben Statements, aber dort führt die ebenfalls ausgeführte Optimierung "Split Life Ranges" eine Umbenennung bei einer der meistbenutzten Variablen durch, die aus ungeklärter Ursache bei der Superblock-DCE nicht durchgeführt wird. Diese Umbenennung verhindert dort anscheinend den WCET-Zuwachs. Solche Interaktionen zwischen verschiedenen Optimierungen sind für die Superblock-DCE nicht erkennbar.

Insgesamt lässt sich aus den Ergebnissen der Superblock-DCE schließen, daß in vielen Benchmarks keine oder nur wenige Superblock-tote Anweisungen vorhanden sind, so daß der Gesamteinfluß der Superblock-DCE unter dem der Superblock-CSE rangiert.

Code Sinking

Als Code Sinking haben wir das Verschieben von lebendigen Anweisungen entlang des Superblocks bezeichnet, wobei die Anweisungen nur in solche Aussprungpunkte kopiert werden, in denen sie lebendig sind. Da diese Optimierung wesentlich aggressiver vorgeht als die reine Superblock-DCE, die nur auf Superblock-totem Code arbeitet, wurde die Möglichkeit geschaffen, das Code Sinking als Erweiterung der Superblock-DCE gesondert zu aktivieren. Die Ergebnisse sind in den Abbildungen 8.12, 8.13 und 8.14 festgehalten, wobei hier nur die Benchmarks aufgeführt sind für die sich eine Änderung im Vergleich zur Standard-Superblock-DCE ergeben hat.

In günstigen Fällen, wie z.B. bei `h264_dec_ldecode_block` werden meist Initialisierungsanweisungen, die nicht in allen folgenden Ausführungspfaden benötigt werden in diejenigen Pfade verschoben, in denen sie effektiv genutzt werden. Die möglichen Vorteile des Code Sinkings lassen sich allerdings genauso schnell in Nachteile verkehren, wenn durch die Verschiebung entlang des Superblocks Anweisungen voneinander getrennt werden, die sich gut für eine gemeinsame Optimierung geeignet hätten, z.B. durch eine lokale Common Subexpression Elimination. Dies hängt stark von der speziellen Situation ab und ist für das Code Sinking selbst, das nur die Abhängigkeiten

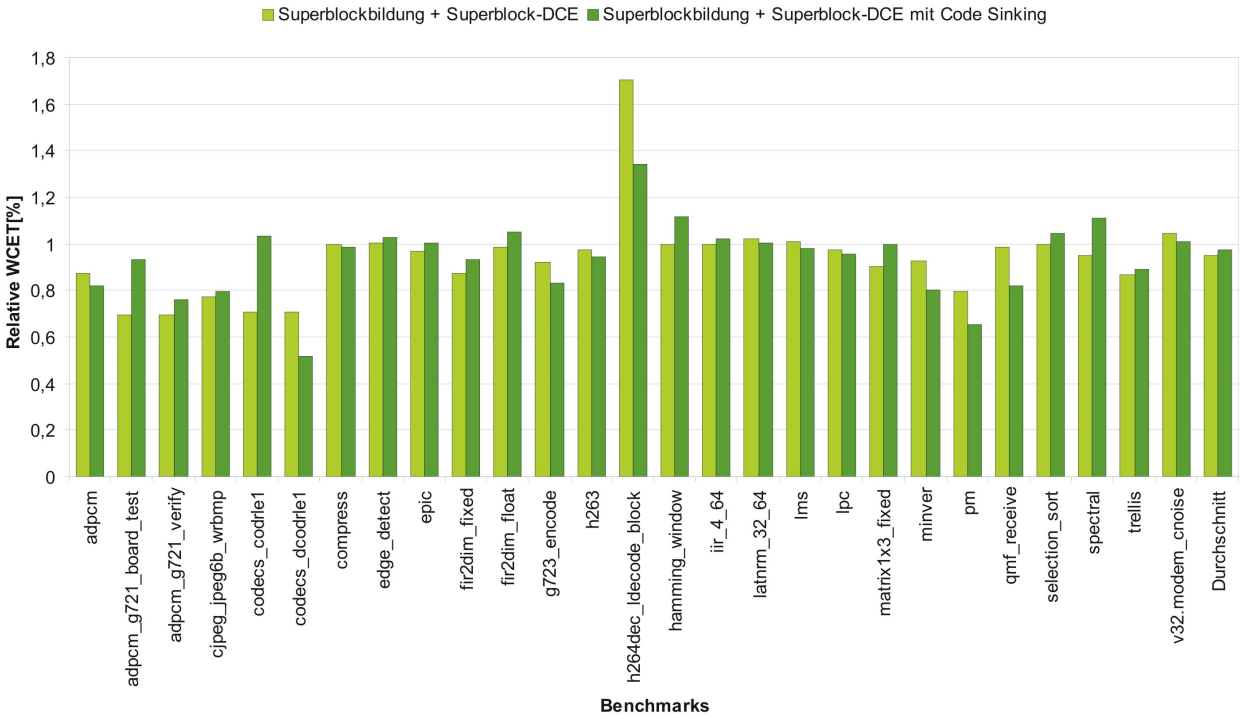


Abbildung 8.12.: WCET-Werte nach Superblock-DCE und Code Sinking (100% ≙ O3 *ohne* ICD-C DCE)

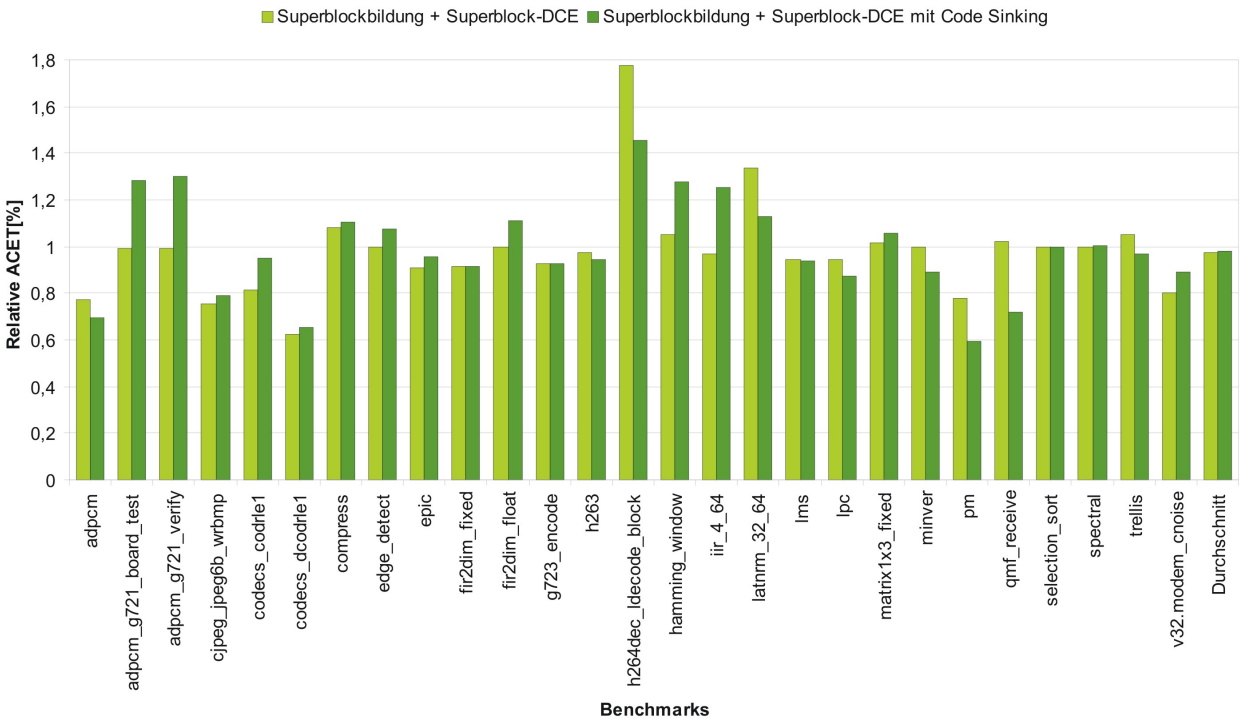


Abbildung 8.13.: ACET-Werte nach Superblock-DCE und Code Sinking (100% ≙ O3 *ohne* ICD-C DCE)

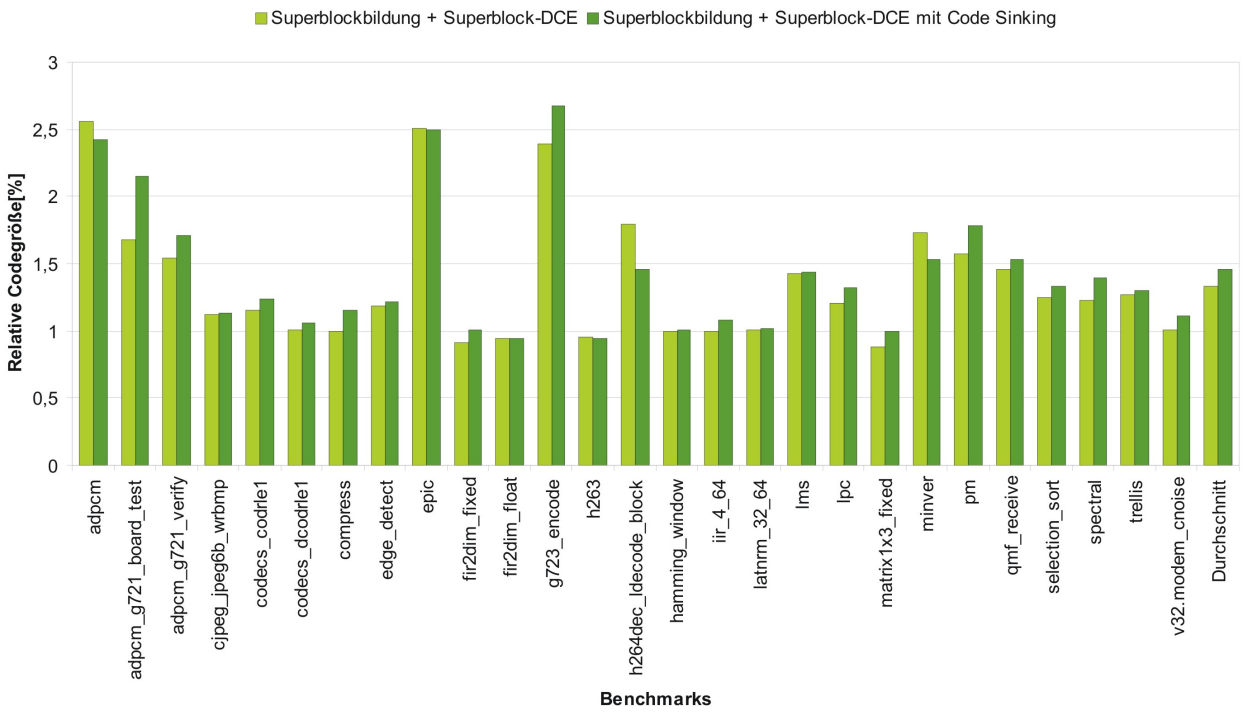


Abbildung 8.14.: Codegrößen nach Superblock-DCE und Code Sinking (100% ≙ O3 *ohne* ICD-C DCE)

zwischen den Anweisungen kennt, nicht beurteilbar. Aus diesem Grund resultiert jedoch für den Durschnitt aller Benchmarks eher eine Verschlechterung der WCET (siehe Tabelle 8.7), auch wenn einzelne Benchmarks durchaus von dieser Optimierung profitieren können, wie in Abbildung 8.12 zu sehen ist.

8.3. Fazit

Im folgenden wollen wir die Ergebnisse dieses Kapitels noch einmal kurz zusammenfassen:

- Die Superblockbildung ist durch die von ihr selbst ausgelösten Effekte und die Synergieeffekte mit anderen Optimierungen in der Lage, die WCET um durchschnittlich 4,83% und die ACET um durchschnittlich 3,24% gegenüber O3 zu verbessern, bei einer Vergrößerung des Codes um durchschnittlich 34% und einer 6,4-fach höheren Kompilierzeit (s. Tabelle 8.1).

- Wir haben festgestellt, daß eine häufigere Nutzung von aiT zur Berechnung des WCEP keine Verbesserung bringt. Daher ist nicht davon auszugehen, daß einzelne WCET-Steigerungen bei der Superblockbildung von nicht beachteten Pfadwechseln ausgehen (s. Tabelle 8.2).

- Die implementierten Erweiterungen `else-if`-Refolding und `switch`-Konvertierung waren in der Lage, gegenüber der normalen Superblockbildung die WCET um durchschnittlich 2,12%, die ACET um durchschnittlich 0,83% und das Codewachstum um durchschnittlich 3% zu senken (s. Tabelle 8.3).

- Das WCET-Optimierungspotential der Standard ICD-C CSE und DCE ist eher gering. Für die CSE ergibt sich im Durchschnitt eine Verschlechterung der WCET um 2,22%, für die DCE eine Verbesserung um 0,03% (s. Tabellen 8.5 und 8.6).

- Im Gegensatz hierzu waren die Superblock-CSE und Superblock-DCE in der Lage, die WCET um durchschnittlich 4,72% bzw. 4,68% zu verbessern. Bei der SB-CSE entstehen 1,3% dieser Verbesserung bereits durch die Superblockbildung selbst, bei der SB-DCE sind dies 2,07%. Insgesamt sind die Superblock-CSE und die Superblock-DCE gegenüber ihren ICD-C Entsprechungen in der Lage, die WCET um durchschnittlich 6,94% bzw. 4,65% zu verbessern (s. Tabellen 8.5 und 8.6).

Es lässt sich also abschließend festhalten, daß sowohl die Superblockbildung selbst, als auch die Erweiterung der Standard-Optimierungen CSE und DCE auf Superblöcke, lohnenswert war, und sich positiv in den Ergebnissen widerspiegelt.

9. Zusammenfassung / Ausblick

Diese Arbeit hat sich erstmals mit der Bildung und Optimierung von Superblöcken zum Zweck der WCET-Minimierung auf der High-Level-Ebene befasst. In Abschnitt 9.1 werden die verwendeten Techniken und die erzielten Ergebnisse noch einmal kurz zusammengefasst, und in Abschnitt 9.2 wird ein kurzer Überblick über mögliche Verbesserungen und Erweiterungen des bestehenden Superblock-Frameworks gegeben.

9.1. Zusammenfassung der Ergebnisse

In dieser Arbeit wurden erstmals High-Level-Formulierungen aller bekannten Superblock-Optimierungen vorgestellt (Abschnitt 4.4), und deren Eignung für die WCET-Optimierung diskutiert. Die vorhandenen Trace-Selektions-Verfahren wurden vorgestellt, und aufgrund ihrer Unzulänglichkeiten in Bezug auf die WCET-Minimierung wurde ein neues Trace-Selektions-Verfahren, das Longest-Path-Verfahren (Abschnitt 6.3), eingeführt.

Wir haben einen Algorithmus entworfen, der die Superblockbildung erstmals auf der High-Level-Darstellung umsetzt (Abschnitt 6.4), der von WCET- und nicht von Profiling-Daten gesteuert wird, und der eine Reihe von Spezialfällen behandelt, die in dieser Form nur auf der High-Level-Ebene auftreten, wie z.B. nicht verschiebbare `break`-Statements. Die Superblockbildung wurde erstmals mit einer Codegrößenkontrolle ausgestattet (Abschnitt 6.3.2), was insbesondere im Kontext der High-Level-Superblöcke notwendig ist, da hier ein höheres Codewachstum als bei den Low-Level-Superblöcken auftritt. Aus diesem Grund haben wir die Konvertierung von `switch`-Anweisungen (Abschnitt 6.4.2) und die Neufaltung von `else-if`-Strukturen (Abschnitt 6.4.3) implementiert, mit deren Hilfe sich bei der Superblockbildung einerseits das Codewachstum reduzieren lässt, und andererseits die WCET reduziert werden kann. Für die Superblockoptimierungen musste außerdem die im WCC vorhandene Back-Annotation um die Behandlung von Kanten-WCECs und Iteration-WCETs ergänzt werden (Abschnitt 3.4.2).

Zur Neuberechnung des WCEP auf der High-Level-Ebene haben wir eine Heuristik auf Basis der Syntaxbaum-basierten WCET-Analyse vorgestellt, mit deren Hilfe die Basisblock-WCETs im Verlauf der Optimierungen effizient aktualisiert werden können (Abschnitt 5.2). Darauf aufbauend wurde eine IPET-basierte Neuberechnung des WCEP, die auf der LLIR-Ebene bereits vorhanden war, auf die ICD-C High-Level-Ebene übertragen, so daß in Zukunft eine effiziente Neuberechnung des WCEP ohne Aufruf von aiT möglich ist (Abschnitt 5.3).

Für die Superblockoptimierungen SB-CSE und SB-DCE wurde die Berechnung von Def/Use-Sets, sowohl in der May- als auch in der Must-Version, implementiert und an die Bedingungen unserer High-Level-IR angepasst (Abschnitt 7.1.3). Die Def/Use-Sets wurden dabei so konzipiert, daß sie mit oder ohne vorhergehende Alias-Analyse benutzbar sind, und im ersteren Fall die Ergebnisse der Alias-Analyse automatisch mit in die Berechnung der Def/Use-Sets einfließen. Dies ermöglicht erstmals eine automatische Nutzung dieser Informationen, die bisher im WCC nur manuell über die explizite Abfrage der Points-To-Sets nutzbar waren. Mithilfe der Def/Use-Sets wurde eine Lebensdauer-Analyse auf Basis des allgemeinen Konzept der Datenflußanalysen entwickelt (Abschnitt 7.1.4), die damit automatisch auch auf die Ergebnisse der Alias-Analyse zurückgreifen kann. Weitere Datenflußanalysen, die für andere Compiler-Optimierungen benutzt werden können,

wie z.B. die Analyse verfügbarer Ausdrücke, sind mit den neuen Def/Use-Sets schnell und einfach umsetzbar.

Zur konkreten Nutzung der berechneten Informationen wurden die beiden wichtigsten Superblockoptimierungen, "Superblock Common Subexpression Elimination" (Abschnitt 7.2) und "Superblock Dead Code Elimination" (Abschnitt 7.3), implementiert. In die Umsetzung der Superblock Dead Code Elimination wurde dabei zusätzlich ein optionales "Code Sinking" integriert.

Die erstellten Optimierungen wurden in Kapitel 8 auf ihre Korrektheit und Leistungsfähigkeit untersucht, wobei für die Superblockbildung eine durchschnittliche WCET-Verbesserung um 4,83%, für die Superblock-CSE eine durchschnittliche WCET-Verbesserung um 6,94% und für die Superblock-DCE eine durchschnittliche WCET-Verbesserung um 4,65% im Vergleich zur Standard-CSE bzw Standard-DCE festgestellt wurde. Hierbei wurde die WCET-gesteuerte Superblockbildung auch zum ersten Mal auf Benchmarks realistischer Größe getestet, die einzige andere Veröffentlichung zum Thema Superblöcke und WCET von Zhao [Zhao et al., 2006], die allerdings Low-Level-Superblöcke betrachtet, hat nur Benchmarks mit durchschnittlich 150 Zeilen untersucht. Für die High-Level-Superblockbildung hat sich hierbei insbesondere die Codegrößenkontrolle als sehr wertvoll erwiesen.

9.2. Ausblick

Der sicherlich interessanteste Punkt zur weiteren Untersuchung der Superblöcke wäre die Umsetzung eines Superblock-basierten Instruction Schedulings im WCC. Hierzu wurde auch bereits eine weitere Diplomarbeit vergeben, die sich mit WCET-gesteuertem Scheduling, das unter anderem mithilfe von Superblöcken realisiert werden soll, befasst. Hierfür können die High-Level-Superblöcke wiederverwendet werden, da ihre Struktur auch in der LLIR-Ebene erhalten bleibt, und dort für das Scheduling benutzt werden kann. Außerdem wäre es möglich die verbleibenden, in Abschnitt 4.4 vorgestellten High-Level Superblockoptimierungen umzusetzen.

Die Aktualisierung der Block-WCET, wie in Abschnitt 5.2 angesprochen, könnte in Zukunft über ein Verfahren des maschinellen Lernens vorgenommen werden, in dem nur die WCET von kompletten Basisblöcken anhand einiger zu extrahierender Merkmale bestimmt wird. Alternativ wäre es auch denkbar hierfür einen Simulator zu benutzen, der die dynamische WCET des Blocks bestimmt. Dies wäre zwar aus den in Abschnitt 2.3 aufgeführten Gründen keine echte WCET, aber eine Garantie kann ohnehin keines dieser heuristischen Verfahren liefern. Da die Verfahren nur benutzt werden, um bestehende WCET-Daten, die von aiT berechnet wurden, zu ergänzen, werden sich einzelne Fehlabschätzungen hier nicht sehr stark auswirken.

In Situationen, in denen keiner der verfügbaren Pfade durch einen Programmteil einen signifikant größeren Teil der WCET verursacht als der Rest der Pfade, könnte es sinnvoll sein High-Level Treegions statt High-Level Superblöcken zu bilden, um keinen der Pfade zu benachteiligen. Eine Treegion ist hierbei ein Baum im CFG, dessen Elemente gleichberechtigt optimiert werden sollen, im Gegensatz zu einem Superblock, bei dem nur die Elemente des Superblock-Pfades gleichberechtigt optimiert werden.

Zur Verbesserung der Laufzeit der Superblock-CSE und Superblock-DCE wäre es sinnvoll die Berechnung der Def/Use-Sets mithilfe von Bitvektoren zu realisieren. Dies ließe sich relativ kurzfristig bewerkstelligen, konnte allerdings innerhalb der Diplomarbeit aus Zeitgründen nicht mehr umgesetzt werden.

Anhang

A. Verwendete Benchmarks

Im folgenden werden die Eigenschaften der verwendeten Benchmarks aus der Testbench des WCC aufgeführt. Die Benchmarks stammen aus verschiedenen bekannten DSP- und Embedded-Testsuiten, wie z.B.

- **DSPstone**: [Živojnović et al., 1994]
- **MediaBench**: [Lee et al., 1997]
- **MiBench**: [Guthaus et al., 2001]
- **MRTC**: [Mälardalen WCET Research Group, 2009]
- **StreamIt**: [Stream It Developers, 2009]
- **UTDSP**: [Lee, 2009]

In den Tabellen ergibt sich die Bedeutung der einzelnen Spalten wie folgt:

A Anzahl Codezeilen

B Anzahl Funktionen

C Anzahl Funktionen mit `goto`s oder unstrukturierten `switch`es

D Anzahl von der Superblockoptimierung ausgeschlossene Funktionen

E Anzahl Basisblöcke

F Anzahl Schleifen

G Anzahl `for`-Schleifen

H Anzahl `if`-Statement

I Anzahl `switch`-Statements

J Durchschnittliche Anzahl `case`s pro `switch`

K Anzahl bedingter Ausdrücke (`&&`, `||`, `?:`)

L Anzahl bedingter Ausdrücke mit bedingten Schreibzugriffen

Benchmark	A	B	C	D	E	F	G	H	I	J	K	L
DSPstone (Fixed Point)												
adpcm_g721_board_test	835	20	0	2	133	14	9	35	0	0	101	2
adpcm_g721_verify	847	20	0	1	135	14	9	36	0	0	101	2
complex_update_fixed	61	2	0	0	2	0	0	0	0	0	0	0
dot_product_fixed	66	2	0	1	5	1	1	0	0	0	0	0
fir2dim_fixed	67	2	0	2	40	13	13	0	0	0	0	0
iir_biquad_N_sections_fixed	97	2	0	2	11	3	3	0	0	0	0	0
matrix1_fixed	61	2	0	2	20	6	6	0	0	0	0	0
n_complex_updates_fixed	76	2	0	2	7	2	2	0	0	0	0	0
real_update_fixed	49	2	0	0	2	0	0	0	0	0	0	0
DSPstone (Floating Point)												
complex_update_float	64	2	0	0	2	0	0	0	0	0	0	0
dot_product_float	69	2	0	1	5	1	1	0	0	0	0	0
fir2dim_float	70	2	0	2	40	13	13	0	0	0	0	0
iir_biquad_N_sections_float	99	2	0	2	11	3	3	0	0	0	0	0
matrix1_float	80	2	0	2	20	6	6	0	0	0	0	0
n_complex_updates_float	77	2	0	2	7	2	2	0	0	0	0	0
n_real_updates_float	61	2	0	2	7	2	2	0	0	0	0	0
real_update_float	54	2	0	0	2	0	0	0	0	0	0	0
MediaBench												
cjpeg_jpeg6b_transupp	3092	6	0	0	164	47	43	15	0	0	0	0
cjpeg_jpeg6b_wrbmp	2243	4	0	0	30	5	5	8	0	0	0	0
epic	1710	7	0	3	141	41	40	9	0	0	18	0
gsm	3024	52	1	14	333	56	50	56	4	4	199	11
gsm_encode	2649	33	1	10	254	48	43	37	4	4	172	10
h264dec_ldecode_block	3549	4	0	1	122	27	26	16	1	5	31	0
h264dec_ldecode_macroblock	1524	2	0	0	114	13	13	35	0	0	76	0
MiBench												
dijkstra	145	5	0	0	32	5	3	9	0	0	1	0
rijndael_decoder	1777	10	0	0	76	10	3	14	2	4	16	1
rijndael_encoder	1827	11	0	0	87	12	4	17	2	4	14	1
sha	639	12	0	2	95	17	7	8	2	9	3	0
misc												
codecs_codrle1	127	6	0	1	33	4	1	10	0	0	5	0
codecs_dcodrle1	75	7	0	2	21	4	3	2	1	3	0	0
g723_encode	1343	26	0	3	201	9	8	55	6	3	30	0
h263	950	5	0	4	23	7	7	0	0	0	14	0
hamming_window	82	1	0	1	10	3	3	0	0	0	0	0
pm	856	14	0	4	151	28	27	27	0	0	3	1
selection_sort	88	2	0	0	11	2	2	2	0	0	2	0

Tabelle A.1.: Eigenschaften der verwendeten Benchmarks (Teil 1).

Benchmark	A	B	C	D	E	F	G	H	I	J	K	L
MRTC												
bsort100	119	3	0	1	18	3	3	3	0	0	0	0
compressdata	309	6	0	2	38	4	2	11	0	0	5	2
cover	257	4	0	0	209	3	3	0	3	64	0	0
expint	98	3	0	0	18	3	3	3	0	0	1	0
fdct	238	2	0	1	7	2	2	0	0	0	0	0
fft1	246	6	0	1	49	11	7	5	0	0	1	0
fir	284	2	0	0	12	2	2	2	0	0	0	0
insertsort	125	1	0	0	7	2	0	0	0	0	0	0
jfdctint	377	2	0	2	10	3	3	0	0	0	0	0
lcdnum	98	2	0	0	24	1	1	1	1	17	0	0
lms	290	8	0	2	49	10	6	5	0	0	0	0
ludcmp	176	3	0	0	46	11	11	5	0	0	1	0
minver	226	4	0	1	73	17	16	10	0	0	5	0
ndes	454	5	0	1	44	12	12	4	0	0	10	0
petrinet	901	1	0	0	81	1	0	52	0	0	72	0
qsort-exam	158	2	0	0	35	6	4	8	0	0	0	0
qurt	186	4	0	0	22	1	1	7	0	0	0	0
select	136	2	0	0	33	4	0	11	0	0	0	0
statemate	1100	8	0	0	226	1	1	82	16	3	80	0
StreamIt												
bitonic	89	4	0	1	17	3	3	3	0	0	0	0
UTDSP												
adpcm	1013	10	0	0	257	8	7	107	0	0	9	0
compress	536	9	0	2	111	22	12	20	1	4	2	0
edge_detect	154	3	0	1	36	10	10	2	0	0	2	0
fft_1024	62	2	0	2	13	4	4	0	0	0	0	0
histogram	47	1	0	1	19	6	6	0	0	0	0	0
iir_4_64	69	2	0	2	10	3	3	0	0	0	0	0
latnrm_32_64	49	2	0	1	10	3	3	0	0	0	0	0
lpc	477	7	0	2	108	23	20	16	0	0	6	0
qmf_receive	87	2	0	0	27	2	1	10	0	0	0	0
spectral	759	10	0	1	169	29	11	38	2	4	2	0
trellis	303	11	0	1	72	10	7	17	0	0	3	0
v32.modem_cnoise	132	2	0	0	12	1	0	3	0	0	2	0

Tabelle A.2.: Eigenschaften der verwendeten Benchmarks (Teil 2).

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[AbsInt, 2009] AbsInt (2009). aiT WCET Analyzer (aiT). http://absint.com.

[Appel & Ginsburg, 1998] Appel, A. W. & Ginsburg, M. (1998). *Modern Compiler Implementation in C*. Press Syndicate of the University of Cambridge.

[Avila et al., 2003] Avila, M., Glaizot, M., & Puaut, I. (2003). Impact of Automatic Gain Time Identification on Tree-Based Static WCET Analysis. In *Proceedings of the 3rd International Workshop on Worst-Case Execution Time Analysis* Porto, Portugal.

[Betts & Bernat, 2006] Betts, A. & Bernat, G. (2006). Tree-Based WCET Analysis on Instrumentation Point Graphs. In *Proceedings of the 9th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*.

[Bihr, 2005] Bihr, H. (2005). Entwicklung einer plattformunabhängigen, kontext-sensitiven Aliasanalyse für das ICD-C Compiler-Framework. Master's thesis, Universität Dortmund, Lehrstuhl Informatik XII.

[Boost, 2009] Boost, C. (2009). Boost Graph Library (BGL). http://www.boost.org/.

[Chakaravarthy & Horwitz, 1986] Chakaravarthy, V. T. & Horwitz, S. (1986). On the Non-Approximability of Points-to Analysis. *Acta Informatica*, 38, 587–598.

[Chang & Hwu, 1988] Chang, P. P. & Hwu, W. W. (1988). Trace Selection for Compiling Large C Application Programs to Microcode. In *Proceedings of the 21st Annual Workshop on Microprogramming and Microarchitecture* (pp. 21–29). Los Alamitos, CA, USA: IEEE Computer Society Press.

[Chang et al., 1991] Chang, P. P., Mahlke, S. A., & Hwu, W. W. (1991). Using Profile Information to Assist Classic Code Optimizations. *Software–Practice and Experience*, 21(12), 1301–1321.

[Chen et al., 2007] Chen, T., Mitra, T., Roychoudhury, A., & Suhendra, V. (2007). Exploiting Branch Constraints without Exhaustive Path Enumeration. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*: Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany.

[Chen et al., 1992] Chen, W. Y., Mahlke, S. A., Warter, N. J., Hank, R. E., Bringmann, R. A., Anik, S., & Hwu, W. W. (1992). Using Profile Information to Assist Advanced Compiler Optimization and Scheduling. *Advances in Languages and Compilers for Parallel Processing*.

[Cohn & Lowney, 2000] Cohn, R. & Lowney, P. G. (2000). Design and Analysis of Profile-Based Optimization in Compaq's Compilation Tools for Alpha. *Journal of Instruction Level Parallelism*, 3, 1–25.

[Colin & Puaut, 2001] Colin, A. & Puaut, I. (2001). A Modular and Retargetable Framework for Tree-based WCET Analysis. In *Proceedings of the 13th Euromicro Conference on Real-Time Systems* (pp. 37–44).

[Cooper et al., 1991] Cooper, K., Hall, M., & Torczon, L. (1991). An Experiment with Inline Substitution. In *Software–Practice and Experience 21(6)* (pp. 581–601).

[Cordes, 2008] Cordes, D. (2008). Schleifenanalyse für einen WCET-optimierenden Compiler basierend auf Abstrakter Interpretation und Polylib. Master's thesis, Technische Universität Dortmund, Lehrstuhl Informatik XII.

[Cousot & Cousot, 1977] Cousot, P. & Cousot, R. (1977). Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *POPL*.

[Davidson & Hollersnm, 1992] Davidson, J. & Hollersnm, A. (1992). Subprogram Inlining: A Study of its Effects on Program Execution Time. *IEEE Transactions on Software Engineering*, 18(2), 89–102.

[Duff, 1983] Duff, T. (1983). Duff's Device. http://de.wikipedia.org/wiki/Duff's_Device.

[Erosa & Hendren, 1994] Erosa, A. & Hendren, L. J. (1994). Taming Control Flow: A Structured Approach to Eliminating Goto Statements. In *Proceedings of 1994 IEEE International Conference on Computer Languages* (pp. 229–240).: IEEE Computer Society Press.

[Esterel, 2009] Esterel (2009). SCADE Suite - The Standard for the Development of Safety-Critical Embedded Software in Aerospace & Defense, Rail Transportation, Energy and Heavy Equipment Industries. http://www.esterel-technologies.com/products/scade-suite/.

[ETAS, 2009] ETAS (2009). ASCET Software Products. http://www.etas.com/en/products/ascet_software_products.php.

[Falk & Kleinsorge, 2009] Falk, H. & Kleinsorge, J. C. (2009). Optimal Static WCET-Aware Scratchpad Allocation. In *Proceedings of the 46th Design Automation Conference*.

[Falk et al., 2006] Falk, H., Lokuciejewski, P., & Theiling, H. (2006). Design of a WCET-Aware C Compiler. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia* Seoul/Korea.

[Ferdinand, 2004] Ferdinand, C. (2004). Worst Case Execution Time Prediction by Static Program Analysis. In *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*.

[Fisher, 1981] Fisher, J. A. (1981). Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, 30(7), 478–490.

[Ford & Fulkerson, 1962] Ford, L. R. & Fulkerson, D. R. (1962). *Flows in Networks*. Princeton University Press.

[Gedikli, 2008] Gedikli, F. (2008). Transformation und Ausnutzung von WCET-Informationen für High-Level-Optimierungen. Master's thesis, Technische Universität Dortmund, Lehrstuhl Informatik XII.

[Ghiya, 1998] Ghiya, R. (1998). *Putting Pointer Analysis to Work*. PhD thesis, School of Computer Science, McGill University, Montreal.

[Guthaus et al., 2001] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., & Brown, R. B. (2001). MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proceedings of the Workload Characterization 2001 IEEE International Workshop* (pp. 3–14). Washington, DC, USA: IEEE Computer Society.

[Hank et al., 1993] Hank, R., Mahlke, S. A., Bringmann, R. A., Gyllenhaal, J. C., & W.Hwu, W. (1993). Superblock Formation Using Static Program Analysis. In *Proceedings of the 26th Annual International Symposium on Microarchitecture* (pp. 247–255).

[Heiko Falk, 2006] Heiko Falk, M. S. (2006). Loop Nest Splitting for WCET-Optimization and Predictability Improvement. In *4th IEEE Workshop on Embedded Systems for Real-Time Multimedia*.

[Horwitz, 1997] Horwitz, S. (1997). Precise Flow-Insensitive May-Alias Analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1), 1–6.

[Hwu et al., 1993] Hwu, W. W., Mahlke, S. A., Chen, W. Y., & et al, P. P. C. (1993). The Superblock: An Effective Technique for VLIW and Superscalar Compilation. *The Journal of Supercomputing*.

[ICD, 2008] ICD (2008). *ICD-C Compiler Framework Developer Manual*. Informatik Centrum Dortmund.

[ICD, 2009] ICD (2009). *ICD Low Level Intermediate Representation Backend Infrastructure (LLIR) – Developer Manual*. Informatik Centrum Dortmund.

[ILOG, 2009] ILOG (2009). CPLEX ILP-solver. http://www.ilog.com/products/cplex/.

[Infineon, 2007] Infineon (2007). *TC1796 32-Bit Single-Chip Microcontroller User's Manual V2.0*. Infineon Technologies AG.

[ISO/IEC, 1999] ISO/IEC (1999). *International Standard 9899 'Programming languages - C'*. ISO/IEC, tc3 edition.

[Kahn, 1962] Kahn, A. B. (1962). Topological Sorting of Large Networks. *Communications of the ACM*, 5(11), 558–562.

[Keller, 2007] Keller, J. (2007). Developers of Real-Time Embedded Software Take Aim at Code Complexity. *Military and Aerospace Electronics*. http://mae.pennnet.com/articles/article_display.cfm?article_id=289158.

[Kidd & Hwu, 2006] Kidd, R. & Hwu, W. M. (2006). Improved Superblock Optimization in GCC. In *GCC Summit*: Center for High Performance and Reliable Computing, University of Illinois at Urbana-Champaign.

[Knoop & Steffen, 1992] Knoop, J. & Steffen, B. (1992). The Interprocedural Coincidence Theorem. In *Proceedings of the 4th International Conference on Compiler Construction* (pp. 125–140).: Springer-Verlag.

[Lee et al., 1997] Lee, C., Potkonjak, M., & Mangione-smith, W. H. (1997). MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *International Symposium on Microarchitecture* (pp. 330–335).

[Lee, 2009] Lee, C. G. (2009). UTDSP Benchmark Suite. http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html.

[Li & Malik, 1995] Li, Y.-T. S. & Malik, S. (1995). Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd ACM/IEEE Design Automation Conference* (pp. 456–461).

[Lim et al., 1995] Lim, S.-S., Bae, Y. H., Jang, G. T., Rhee, B.-D., Min, S. L., Park, C. Y., Shin, H., Park, K., Moon, S.-M., & Kim, C. S. (1995). An Accurate Worst Case Timing Analysis for RISC Processors. *IEEE Transactions on Software Engineering*, 21(7), 593–604.

[Lokuciejewski, 2007] Lokuciejewski, P. (2007). *A WCET-Aware Compiler. Design, Concepts and Realization*. Vdm Verlag Dr. Müller.

[Lokuciejewski et al., 2009a] Lokuciejewski, P., Cordes, D., Falk, H., & Marwedel, P. (2009a). A Fast and Precise Static Loop Analysis based on Abstract Interpretation, Program Slicing and Polytope Models. In *International Symposium on Code Generation and Optimization (CGO)* Seattle / USA.

[Lokuciejewski et al., 2008] Lokuciejewski, P., Falk, H., Marwedel, P., & Theiling, H. (2008). WCET-Driven, Code-Size Critical Procedure Cloning. In *Proceedings of 11th International Workshop on Software & Compilers for Embedded Systems.*

[Lokuciejewski et al., 2009b] Lokuciejewski, P., Gedikli, F., & Marwedel, P. (2009b). Accelerating WCET-Driven Optimizations by the Invariant Path Paradigm: A Case Study of Loop Unswitching. In *Proceedings of the 12th International Workshop on Software and Compilers for Embedded Systems* (pp. 11–20).

[Lokuciejewski et al., 2009c] Lokuciejewski, P., Gedikli, F., Marwedel, P., & Morik, K. (2009c). Automatic WCET Reduction by Machine Learning Based Heuristics for Function Inlining. In *Proceedings of The 3rd Workshop on Statistical and Machine Learning Approaches to Architecture and Compilation.*

[Lokuciejewski & Marwedel, 2009] Lokuciejewski, P. & Marwedel, P. (2009). Combining Worst-Case Timing Models, Loop Unrolling, and Static Loop Analysis for WCET Minimization. *Proceedings of 21st Euromicro Conference on Real-Time Systems*, (pp. 35–44).

[Lowney et al., 1993] Lowney, G. P., Freudenberger, S. M., Karzes, T. J., Lichtenstein, W. D., Nix, R. P., O'Donnell, J. S., & Ruttenberg, J. C. (1993). The Multiflow Trace Scheduling Compiler. *The Journal of Supercomputing*, 7(1-2), 51–142.

[lp_solve, 2009] lp_solve, C. (2009). `lp_solve` - Mixed Integer Linear Programming (MILP) solver. http://lpsolve.sourceforge.net/.

[Marwedel, 2007] Marwedel, P. (2007). *Eingebettete Systeme.* Morgan Kaufmann.

[Mälardalen WCET Research Group, 2009] Mälardalen WCET Research Group (2009). MRTC Benchmarks. http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.

[Muchnick, 1997] Muchnick, S. S. (1997). *Advanced Compiler Design and Implementation.* Morgan Kaufmann.

[Park & Shaw, 1991] Park, C. Y. & Shaw, A. C. (1991). Experiments with a Program Timing Tool Based on Source-Level Timing Schema. *Computer*, 24(5), 48–57.

[P.F. Elzer, 2006] P.F. Elzer, H. Chen, V. N. (2006). "Mixed Reality"- kann der Computer zum echten Helfer werden? *Informatik aktuell - Echtzeitsysteme im Alltag*, (pp. 81–90).

[Prasad et al., 2004] Prasad, W. Z., Kulkarni, P., Whalley, D., Healy, C., Mueller, F., & ryung Uh, G. (2004). Tuning the WCET of Embedded Applications. In *10th IEEE Real-Time and Embedded Technology and Applications Symposium* (pp. 472–481).

[Puaut, 2006] Puaut, I. (2006). WCET-Centric Software-controlled Instruction Caches for Hard Real-Time Systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems* (pp. 217–226). Washington, DC, USA: IEEE Computer Society.

[Ramalingam, 1994] Ramalingam, G. (1994). The Undecidability of Aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5), 1467–1471.

[Rotthowe, 2008] Rotthowe, F. (2008). Scratchpad-Allokation von Daten zur Worst-Case Execution Time Minimierung. Master's thesis, Technische Universität Dortmund, Lehrstuhl Informatik XII.

[Rüthing, 2008] Rüthing, O. (2008). *Begleitmaterial zur Vorlesung 'Formale Methoden des Systementwurfs' im SS2008.* Technical report, Technische Universität Dortmund, Lehrstuhl Informatik V.

[Schmoll, 2008] Schmoll, F. (2008). ILP-basierte Registerallokation unter Ausnutzung von WCET-Daten. Master's thesis, Technische Universität Dortmund, Lehrstuhl Informatik XII.

[Schulte, 2007] Schulte, D. (2007). Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler. Master's thesis, Technische Universität Dortmund, Lehrstuhl Informatik XII.

[Sieber, 2006] Sieber, A. (2006). DVB-T-C-S Digitales Fernsehen über Antenne, Kabel und Satellit, MHP Echtzeitübertragung im digitalen Fernsehen. *Informatik aktuell - Echtzeitsysteme im Alltag*, (pp. 11–15).

[Steinle & Halang, 2006] Steinle, E. & Halang, W. A. (2006). Ein PEARL-Einplatinenrechner hält Einzug auf dem Bauernhof. *Informatik aktuell - Echtzeitsysteme im Alltag*, (pp. 101–108).

[Stream It Developers, 2009] Stream It Developers (2009). Stream It Testbench. http://groups.csail.mit.edu/cag/streamit/shtml/benchmarks.shtml.

[Su et al., 1984] Su, B., Ding, S., & Jin, L. (1984). An Improvement of Trace Scheduling for Global Microcode Compaction. *SIGMICRO Newsletter*, 15(4), 78–85.

[Symtavision, 2009] Symtavision (2009). SymTA/S. http://www.symtavision.com.

[Telelogic, 2009] Telelogic (2009). Statemate. http://www.telelogic.com/products/statemate/index.cfm.

[Tonella, 1999] Tonella, P. (1999). Effects of Different Flow Insensitive Points-to Analyses on DEF/USE Sets. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering* (pp.62). Washington, DC, USA: IEEE Computer Society.

[Živojnović et al., 1994] Živojnović, V., Velarde, J. M., Schläger, C., & Meyr, H. (1994). DSPSTONE: A DSP-oriented Benchmarking Methodology. In *Proceedings of the International Conference on Signal Processing and Technology.*

[Wegener, 1999] Wegener, I. (1999). *Theoretische Informatik - Eine algorithmenorientierte Einführung*, volume 2. B. G. Teubner, Stuttgart Leipzig.

[Weiser, 1991] Weiser, M. (1991). The Computer for the Twenty-First Century. *Scientific American*, (pp. 94–10).

[Wilhelm et al., 2008] Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., & Stenström, P. (2008). The Worst-Case Execution-Time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 1–47.

[Zöbel et al., 2006] Zöbel, D., Berg, U., & Schönfeld, M. (2006). Visuelle Lenkassistenz für Fahrzeuge mit Einachsanhänger. *Informatik aktuell - Echtzeitsysteme im Alltag*, (pp. 1–10).

[Zhao et al., 2006] Zhao, W., Kreahling, W., Whalley, D., Healy, C., & Mueller, F. (2006). Improving WCET by Applying Worst-Case Path Optimizations. *Real-Time Systems*, 34(2), 129–152.