UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

Daniel Schulte

Modellierung und Transformation von Flow Facts in einem WCET-optimierenden Compiler

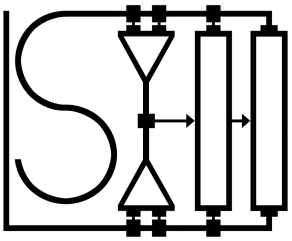
Diplomarbeit

8. Nov. 2006 – 4. Mai 2007

INTERNE BERICHTE
INTERNAL REPORTS

Lehrstuhl 12 für Eingebettete Systeme
Fachbereich Informatik
Universität Dortmund

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



GERMANY · D-44221 DORTMUND

Daniel Schulte, Königsberger Str. 8, 58675 Hemer
Matr.-Nr. 96782, Fachbereich Informatik, Universität Dortmund

In den vergangenen Monaten durfte ich einen erheblichen Teil meiner Zeit in diese Diplomarbeit investieren. Dementsprechend mussten viele Bekannte und Freunde, aber auch z. T. meine Familie bei verschiedenen Gelegenheiten auf mich verzichten. Auch bei einigen Projekten, die ich sonst gern mit meiner ehrenamtlichen Tätigkeit unterstütze, musste ich kürzer treten. Ich möchte mich bei allen für ihre Geduld und ihr Verständnis, aber auch für ihren Zuspruch bedanken.

Ganz besonderen Dank möchte ich aber an jene richten, die mich aktiv beim Erstellen dieser Arbeit unterstützt haben. Die Mitarbeiter am Lehrstuhl, insbesondere Dr. Heiko Falk, nahmen sich stets für meine Fragen und für anregende Gespräche Zeit. Verschiedene Ideen durfte ich in Diskussionen mit meine Komolitonen, insbesondere mit Judith Ackermann, Michael Poch und Harald Günther, festigen. Die gemeinsamen Experimente beim Erstellen von Graphiken in Latex mit Daniel Höcker waren gewinnbringend und unterhaltsam.

Auch meinem Eltern danke ich für ihre Unterstützung und ihr Interesse an meiner Arbeit.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einführung

Dieses Kapitel wird in drei Schritten zur Diplomarbeit hinführen: Die Motivation in Abschnitt 1.1 wird den Kontext der Diplomarbeit aufzeigen, deren Ziele werden in Abschnitt 1.2 benannt. Abschnitt 1.3 stellt den Aufbau dieser Diplomarbeit vor.

1.1 Motivation

Digitale Systeme sind in alle Bereiche des modernen Alltags eingezogen. Nicht nur, dass viele Menschen in Deutschland an einem Büroarbeitsplatz mit Computern arbeiten, auch in vielen deutschen Haushalten wird ein PC für verschiedenste Aufgaben eingesetzt. So nutzten im Jahr 2005 insgesamt 70% der Deutschen einen Computer, 2/3 davon täglich [Statistisches Bundesamt 2006].

Digitale Systeme haben aber nicht nur in Form von Computern alle Lebensbereiche durchdrungen, vielmehr noch gelang dies den **Eingebetteten Systemen**. Das sind „informationsverarbeitende Systeme, die in ein größeres, umgebendes System eingebettet sind“ [Marwedel 2001].

Von der Intelligent Sensor Technik eines Toasters, über das Mobiltelefon und den MP3-Player bis hin zum Auto kommt jeder Industriestaaten-Bewohner heute täglich mit einer Vielzahl Eingebetteter Systeme in Kontakt. Schlagworte wie *disappearing computer* und *ubiquitous computing* beschreiben dieses dritte Zeitalter der Informatik [Marwedel 2006].

Selbst in einem Kleinwagen verrichten heute 20 Steuergeräte ihren Dienst, in einem Oberklassefahrzeug sind es bis zu 70 Steuereinheiten für Motorregelung, Stoßdämpferabstimmung, Fensterheber oder Klimaautomatik. Aber nicht nur für Komfortsysteme werden die digitalen Steuergeräte in Fahrzeugen eingesetzt. Insbesondere der Sicherheit des Fahrens galt die Aufmerksamkeit der Entwickler: ABS, ESP, automatisches Einparken, Spurhalte- und Spurwechelassistent wären ohne Controller nicht realisierbar. Und erst recht künftige Techniken wie eine Stop-and-Go-Automatik, bei der in Stausituationen dem vorausfahrenden Fahrzeug automatisch gefolgt werden soll, werden intensiv die Möglichkeiten digitaler Steuersysteme nutzen [Welt.de 2006].

Die Anforderungen an diese Systeme unterscheiden sich durch ihre Verwendung wesentlich von denen eines klassischen PCs. Sie müssen klein, leicht, robust und energiesparsam, darüber hinaus aber auch wirtschaftlich rentabel für den Einsatz in alltäglichen Produkten sein, um einige Ziele zu nennen. Dabei widersprechen sich diese Ziele zum Teil. Erstere erfordern z. B. stark spezialisierte Prozessoren, um jede nicht benötigte Funktionalität und damit Fläche und Energie einzusparen. Produktions- und Entwicklungskosten dagegen lassen sich oft

durch multipurpose Prozessoren erheblich senken. Zwischen diesen divergierenden Zielen ist stets projektspezifisch ein Ausgleich zu finden. Dazu gehört die *Hardware/Software Partitionierung*, also die Entscheidung, welche Funktionalitäten in Hardware und welche in Software zu realisieren sind, aber z. B. auch die Entscheidung, mit welcher Taktfrequenz ein System zu betreiben ist. Die Zusammenhänge zwischen den einzelnen Faktoren werden in [Marwedel 2006] genauer untersucht.

Neben den bisher genannten Größen ist in Eingebetteten Systemen insbesondere die Reaktionszeit oft von besonderer Bedeutung. Eine zu späte Berechnung eines Ergebnisses ist mit einem fehlerhaften Ergebnis gleich zu setzen. So kann eine zu späte Reaktion eines ESPs nicht nur nutzlos sein, sondern gar die Situation des Autofahrers verschlimmern. Ein solches Fehlschlagen ist auszuschließen.

Systeme, bei denen eine verspätete Berechnung eines Ergebnisses wie beim ESP zu einer Katastrophe führen kann, die mitunter sogar Menschenleben in Gefahr bringt, werden **harte Realzeitsysteme** genannt. Aber nicht nur in diesen, sondern auch in vielen sogenannten **weichen Realzeitsystemen** ist heute eine garantierte maximale Bearbeitungsdauer notwendig. So sind zwar kleine Aussetzer bei der Wiedergabe einer DVD keine Katastrophe, aber bei einem hochwertigen Markengerät rufschädigend für den Hersteller.

Um die Reaktionszeit eines Systems mit seinen Anforderungen vergleichen zu können, ist Wissen über dessen **WCET** (*worst case execution time* = längst mögliche Programmausführungzeit) notwendig. Da diese jedoch i. d. R. nicht zur Verfügung steht und auch nicht ohne weiteres ermittelbar ist, wird gerade in harten Realzeitsystemen Hardware mit hoher Rechenleistung eingesetzt. Dies geschieht in der Hoffnung, so stets rechtzeitig Ergebnisse zu erhalten, und geht u. a. zu Lasten der Kosten und des Energieverbrauchs eines Produkts. Zudem kann eine rechtzeitige Reaktion immer noch nicht garantiert werden.

Dieser Aspekt insbesondere Eingebetteter Systeme rückte in den letzten Jahren in den Mittelpunkt einiger Forschungseinrichtungen und Firmen. So existieren mittlerweile einige wenige Anwendungen, die anhand eines Programms und zusätzlich benötigten Informationen einen Schätzwert für dessen WCET ermitteln können.

Auch für den Lehrstuhl 12 für Eingebettete Systeme an der Universität Dortmund ist die WCET von Programmen Teil seiner Forschungstätigkeiten. Schwerpunkt ist aber nicht die Berechnung der WCET, sondern die Integration dieser Berechnung in einen Compiler. Dies ermöglicht, die WCET automatisch während des Compiliervorgangs zu ermitteln, die Menge der zusätzlich benötigten Informationen zu minimieren und Programme bzgl. ihrer WCET zu optimieren.

1.2 Ziel der Diplomarbeit

Im Rahmen der Forschungstätigkeit am Lehrstuhl 12 der Universität Dortmund entsteht für den Infineon TriCore Prozessor der C Compiler **WCC** (WCET-Aware C Compiler), in den eine Analyse der WCET bereits integriert ist [Falk u. a. 2006; Lokuciejewski 2005]. Für diese Analyse wird das Tool *aiT* der Firma AbsInt Angewandte Informatik GmbH genutzt [AbsInt 2006]. Neben einem zu analysierenden Programm benötigt das Tool zusätzliche Informationen

zu diesem, insbesondere Kontrollflussinformationen, um eine möglichst präzise Abschätzung der WCET zu berechnen.

Im WCC wird jedes zu compilierende Programm auf verschiedenen Abstraktionsebenen dargestellt, die jeweils individuelle Analyse- und Optimierungstechniken unterstützen. Bisher werden die Kontrollflussinformationen zur WCET-Berechnung für ein Programm auf der niedrigsten Abstraktionsebene, das zugehörige Programm selbst aber auf der höchsten Abstraktionsebene – nämlich durch seinen Quellcode – gegeben. Dies ist insbesondere daher problematisch, da durch die Übersetzungen und Optimierungen des Compilers der Kontrollfluss eines Programms modifiziert werden kann. So ist es für einen Programmierer schwer, Kontrollflussinformationen zu dem Quellcode eines Programms auf niedriger Abstraktionsebene korrekt zu spezifizieren.

Ziel der Diplomarbeit ist daher, die zusätzlich benötigten Kontrollflussinformationen zur WCET-Berechnung für eine Darstellung auf hoher Abstraktionsebene zu modellieren, sowie Techniken zur Konsistenzsicherung dieser Informationen für die Übersetzungen und Optimierungen des Programms im Compiler zu entwickeln.

Letztendlich sind die im Quellcode eines Programms spezifizierten Kontrollflussinformationen zur WCET-Analyse an das Tool aiT zu übergeben.

1.3 Aufbau der Diplomarbeit

Die Diplomarbeit gliedert sich wie folgt:

Kapitel 2 gibt eine kurze Einführung in die WCET-Analyse und stellt das Programm aiT von AbsInt vor. Insbesondere werden die von diesem Programm benötigten zusätzlichen Informationen betrachtet.

Kapitel 3 stellt den WCC und dessen Komponenten vor. Der Umgang mit den zusätzlichen Informationen für die WCET-Analyse im WCC vor der Diplomarbeit wird erläutert. Auf der Beschreibung von aiT und dem WCC aufbauend werden die Ziele der Diplomarbeit aus Abschnitt 1.2 konkretisiert, sowie die notwendigen Änderungen am WCC aufgezeigt.

Kapitel 4 führt den Begriff der Flow Facts ein. Dazu gehört ihre mathematische Modellierung, die Entwicklung geeigneter Datenstrukturen sowie ein Interface zu deren Annotierung im Quellcode eines zu analysierenden Programms.

Kapitel 5 umfasst den Schwerpunkt der Diplomarbeit. Die Transformationen eines zu analysierenden Programms im WCC werden klassifiziert und Mechanismen, um auch Flow Facts entsprechend zu transformieren, werden vorgestellt. Dabei wird insbesondere die Sicherheit und Präzision dieser Transformationen von Bedeutung sein.

Kapitel 6 demonstriert anhand praktischer Ergebnisse die Korrektheit des Ansatzes.

Kapitel 7 fasst die Ergebnisse dieser Arbeit kurz zusammen und gibt einen Ausblick auf Erweiterungen vorgestellter Mechanismen für Flow Facts und auf zukünftige Entwicklungsmöglichkeiten, die u. a. durch die Integration der Flow Facts in den WCC geschaffen wurden.

2 WCET-Analyse

Wie im ersten Kapitel bereits dargestellt wurde, ist die WCET für Eingebettete Systeme eine wichtige Größe. Vor allem in zweierlei Hinsicht ist ihre Kenntnis bedeutend:

1. **Sicherheit**
   Da Eingebettete Systeme oft in sicherheitskritischen Anwendungen eingesetzt werden, müssen sie u. a. zuverlässig sein [Marwedel 2006]. Ein Aspekt dieser Zuverlässigkeit ist die Sicherheit, dass ein fehlschlagendes System keinen Schaden verursacht. Aber gerade bei harten Realzeitsystemen entspricht eine verspätete Berechnung des Ergebnisses einem Fehlschlag und kann in einer Katastrophe enden. Bei diesen Systemen muss also garantiert werden, dass sie stets rechtzeitig reagieren. Diese Garantie ist mit der Kenntnis der WCET eines Programms möglich.
2. **Wirtschaftlichkeit**
   Ist die WCET eines Programms für verschiedene Systeme bekannt, so kann aus den Systemen das Rentabelste für die Unternehmung ausgewählt werden. In diese Auswahl können nicht nur wirtschaftliche Faktoren sondern auch weitere Faktoren wie Energie- und Platzverbrauch einfließen.

Dieses Kapitel wird die WCET-Analyse erläutern. Dazu werden in Abschnitt 2.1 grundlegende Begriffe definiert. Abschnitt 2.2 zeigt in die Möglichkeiten zur Berechnung einer WCET auf, und in Abschnitt 2.3 wird das Programm aiT, das im WCC genutzt wird, vorgestellt.

In Abschnitt 2.4 werden abschließend die zusätzlich benötigten Informationen zur WCET-Analyse untersucht.

2.1 Kontrollfluss und WCET

Die Ausführungszeit eines Programms (auf einer bestimmten Hardware) wird vor allem durch dessen Kontrollfluss bestimmt. Unter Kontrollfluss ist die Abarbeitungsreihenfolge von Anweisungen zu verstehen. Diese wird durch Kontrollstrukturen wie z. B. Schleifen oder bedingte Ausführungen beeinflusst.

Bei der Betrachtung von Kontrollflusspfaden wird i. d. R. von den einzelnen Anweisungen abstrahiert und statt dessen der Kontrollfluss zwischen Basisblöcken betrachtet.

Definition 2.1
Ein **Basisblock** ist eine maximale Sequenz von Anweisungen, die nur über die erste Anweisung betreten und über die letzte Anweisung verlassen werden kann [nach Muchnick 1997].

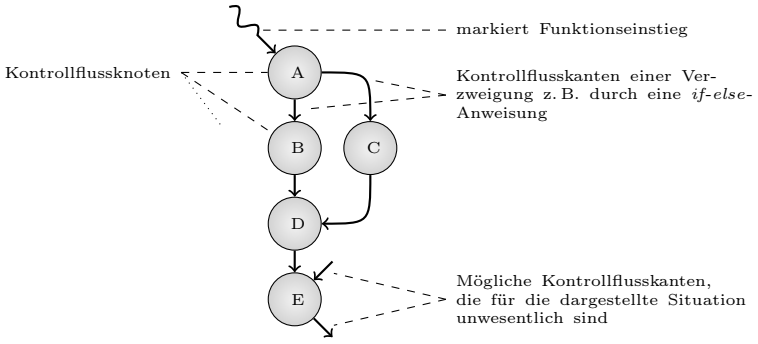


Abbildung 2.1: Bsp. eines Kontrollflussgraphen

Basisblöcke sind die Knoten im Kontrollflussgraph (vgl. Abb. 2.1), gerichtete Kanten die durch die Kontrollstrukturen bestimmten Vorgänger-Nachfolger Relationen im Programmfluss. Die Länge eines Pfades wird als Summe der Ausführungszeiten der Basisblöcke entlang des Pfades gemessen.

Wegen der Kontrollstrukturen, die durch Parameter beeinflusst werden können, gibt es für ein Programm in der Regel nicht einen Kontrollflusspfad, sondern eine Menge dieser Pfade.

Definition 2.2
Die **Menge aller möglichen Kontrollflusspfade** eines Programms $\mathcal{P}$ heißt $CFP(\mathcal{P})$ (Control Flow Paths)[nach Kirner 2003].

Dabei heißt „möglich“ nicht, dass dieser Pfad in der Ausführung auch wirklich genommen werden kann, sondern nur, dass er durch die Syntax des Programms erlaubt ist. Folglich ist $CFP(\mathcal{P})$ für ein Programm mit einer Schleife unbegrenzt, auch wenn das Programm stets terminiert, da die Auswertung der Abbruchbedingung der Schleife für die Menge $CFP(\mathcal{P})$ keine Rolle spielt.

Unter $CFP_{opt}(\mathcal{P})$ ist die Kontrollflusspfadmenge zu verstehen, die ein allwissender Beobachter sehen könnte. Die Länge des längsten Pfades $CFP_{WCET,opt}(\mathcal{P})$ in ihr ist dementsprechend die längste mögliche Berechnungszeit eines Programms $\mathcal{P}$, WCET genannt. Allerdings ist $CFP_{opt}(\mathcal{P})$ (und damit auch $CFP_{WCET,opt}(\mathcal{P})$) wegen der Unentscheidbarkeit der WCET-Analyse i. d. R. weder bekannt noch effizient berechenbar (siehe Abschnitt 2.2 auf der nächsten Seite).

Definition 2.3
Die **WCET** (*worst case execution time)* eines Programms $\mathcal{P}$ auf einem System $\mathcal{S}$ ist dessen längste mögliche Ausführungszeit auf diesem bestimmten System unter der Voraussetzung, dass das Programm $\mathcal{P}$ isoliert auf diesem läuft [nach Ermedahl 2003].

Isoliert bedeutet in diesem Zusammenhang, dass ein Programm allein auf der Zielhardware – insbesondere also auch ohne Betriebssystem – läuft, seine Ausführung nicht (z. B. durch Interrupts) unterbrochen wird, und Hintergrundaktivitäten wie *DMA* (*direct memory access*) oder das Auffrischen von DRAM-Speicherzellen nicht berücksichtigt werden.

Analog zur WCET lassen sich auch die *BCET* (*best case execution time*) als die kürzeste mögliche und die *ACET* (*average case execution time*) als die durchschnittliche Ausführungszeit eines Programms $\mathcal{P}$ auf einem System $\mathcal{S}$ definieren.

Für eine Analyse der WCET lässt sich die Menge $CFP(\mathcal{P})$ durch zusätzliche Informationen *info* einschränken, um so möglichst genau die Menge $CFP_{opt}(\mathcal{P})$ zu beschreiben.

$$CFP_{info}(\mathcal{P}) \subseteq CFP(\mathcal{P}) \tag{2.1}$$

Dabei ist zu beachten, dass die Einschränkung von $CFP(\mathcal{P})$ zur Beschreibung von $CFP_{opt}(\mathcal{P})$ sicher ist.

$$CFP_{opt}(\mathcal{P}) \subseteq CFP_{info}(\mathcal{P}) \tag{2.2}$$

Denn damit gilt insbesondere

$$CFP_{WCET,opt}(\mathcal{P}) \in CFP_{info}(\mathcal{P}), \tag{2.3}$$

und das Ergebnis einer auf $CFP_{info}(\mathcal{P})$ basierenden WCET-Analyse erlaubt eine Aussage zur Sicherheit eines Systems.

Die Begriffe ausführbare und unausführbare Pfade bezeichnen das Verhältnis eines Pfades zu $CFP_{opt}(\mathcal{P})$.

Definition 2.4
Ein Kontrollflusspfad p für ein Programm $\mathcal{P}$ wird **ausführbar** genannt, wenn er tatsächlich während der Programmausführung genommen werden kann, also wenn $p \in CFP_{opt}(\mathcal{P})$ gilt. Andernfalls wird er **unausführbar** genannt [nach Kirner 2003].

Auf diesen Grundlagen basierend kann die Analyse der WCET eingeführt werden.

2.2 Berechnung der WCET

Das Problem „Berechnung der WCET eines Programms $\mathcal{P}$“ lässt sich in das *Halteproblem* überführen. Denn wenn die WCET eines jeden Programms effizient berechnet werden könnte, dann ließe sich insbesondere auch bestimmen, ob dieses Programm terminiert. Allerdings bewies Alan Turing schon 1936, dass das Halteproblem nicht entscheidbar ist. Folglich ist die Berechnung der WCET ebenfalls nicht entscheidbar. Die WCET eines Programms kann daher nur abgeschätzt werden [Kirner u. Puschner 2005b].

Im Folgenden steht $\mathrm{WCET}_{\mathrm{est}}$ für eine abgeschätzte WCET, $\mathrm{WCET}_{\mathrm{real}}$ hingegen für die wahre, i. d. R. unbekannte WCET (vgl. Abb. 2.2 auf der nächsten Seite).

In Abschnitt 2.2.1 werden die Anforderungen an eine Approximation der WCET analysiert, während in Abschnitt 2.2.2 die beiden etablierten Ansätze derer Approximation vorgestellt werden.

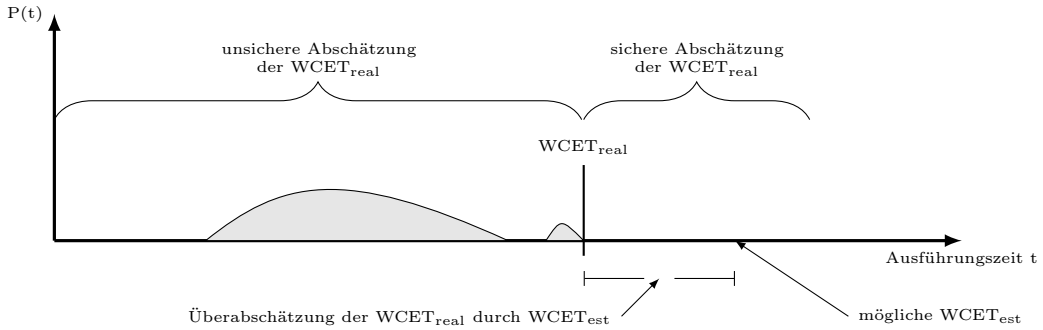


Abbildung 2.2: Abschätzung der WCET

2.2.1 Anforderungen an die Approximation

Der wichtigste Einsatzzweck der WCET ist die Garantie der Sicherheit von Eingebetteten Systemen. Eine approximierte $\mathrm{WCET}_{\mathrm{est}}$ muss stets größer gleich der unbekannten tatsächlichen $\mathrm{WCET}_{\mathrm{real}}$ sein, um nicht irrtümlich wegen einer zu kleinen $\mathrm{WCET}_{\mathrm{est}}$ zu garantieren, dass ein Programm alle vorgegebenen Zeitschranken einhalten wird.

Definition 2.5
Kann für eine Approximation $\mathrm{WCET}_{\mathrm{est}}$ garantiert werden, dass

$$\mathrm{WCET}_{\mathrm{est}} \geq \mathrm{WCET}_{\mathrm{real}} \tag{2.4}$$

gilt, so wird diese Approximation der $\mathrm{WCET}_{\mathrm{real}}$ **sicher** genannt.

Neben der Sicherheit ist aber auch zu gewährleisten, dass die Überabschätzung der $\mathrm{WCET}_{\mathrm{real}}$ möglichst gering ist, also dass

$$\mathrm{WCET}_{\mathrm{est}} - \mathrm{WCET}_{\mathrm{real}} \rightarrow min \tag{2.5}$$

gilt. Ansonsten sind die Ergebnisse für den praktischen Einsatz nicht brauchbar. Künftige Optimierungstechniken im WCC mit dem Ziel der $\mathrm{WCET}_{\mathrm{real}}$-Reduktion wären durch die $\mathrm{WCET}_{\mathrm{est}}$ nicht hinreichend bewertbar, eine Auswahl von Hardware anhand der $\mathrm{WCET}_{\mathrm{est}}$ durch eine Unternehmung wirtschaftlich nicht sinnvoll.

Folglich ist stets zu beachten, dass notwendige Approximationen **sicher** aber gleichzeitig möglichst **präzise** sind.

2.2.2 Techniken zur Approximation

Zwei Techniken der Approximation der $\mathrm{WCET}_{\mathrm{real}}$ eines Programms $\mathcal{P}$ für ein System $\mathcal{S}$ haben sich etabliert.

Statische WCET-Analyse

Bei der statischen Analyse basiert die Berechnung der $WCET_{est}$ auf einem *Timing Model* der Zielhardware und dem Assemblercode des zu analysierenden Programms. Wegen der Unentscheidbarkeit der WCET-Berechnung sind dabei i. d. R. zusätzliche Informationen über den Kontrollfluss des zu analysierenden Programms notwendig (vgl. Abb. 2.3).

```
int square ( int i ){
  return i*i;
}
```

Sequentieller Code, keine Kontrollflussinformationen notwendig.

```
int facult ( int i ){
  int result = 1;
  for( ; i > 0; i -= 1 ){
    result *= i;
  }
  return result;
}
```

Benötigt Kontrollflussinformationen. Weiß z. B. der Programmierer, dass höchstens die Fakultät von 100 berechnet werden muss – größere Eingaben werden vielleicht anderweitig abgefangen – so kann er „Schleife wird max 100 mal iteriert“ annotieren.

Abbildung 2.3: Bsp. für notwendige Kontrollflussinformationen bei statischer Analyse

Die statische Analyse berechnet eine sichere und präzise untere Schranke für die $WCET_{real}$ und ist selbst Teil aktueller Forschungen und Entwicklungen [Puschner u. Burns 2000; Holsti u. Saarinen 2002; Heckmann u. Ferdinand 2004].

Dynamische WCET-Analyse

Wegen fehlender theoretischer Grundlagen für eine statische Analyse basieren die bisherigen Techniken der dynamischen WCET-Analyse auf Testläufen des zu analysierenden Programms auf der Zielhardware. Da das Testen aller Fälle i. d. R. nicht in endlicher Zeit möglich ist, ermittelt die dynamische Analyse eine nicht sichere Abschätzung der WCET. Techniken wie die *genetische Programmierung* verbessern zwar die Qualität der Ergebnisse, dennoch kann die Sicherheit der Abschätzung nicht garantiert werden.

WCET Analyse für den WCC

Ohne weitere Vor- und Nachteile zu betrachten, erfordert die Sicherheit Eingebetteter Systeme eine statische WCET-Analyse, auch wenn diese Kontrollflussinformationen und ein aufwändiges Modell der Zielplattform benötigt.

2.3 Tool zur Berechnung der $WCET_{est}$

Am Lehrstuhl 12 der Universität Dortmund kommt im WCC das statische WCET-Analyse-Tool aiT der Firma AbsInt zum Einsatz [Falk u. a. 2006]. Dieses wird in Abschnitt 2.3.1 vorgestellt. Besonderes Augenmerk gilt den zusätzlichen Informationen zur WCET-Analyse für aiT – spezifiziert in einer Datei im sogenannten *AIS-Format* – im Abschnitt 2.3.2.

Abschließend wird in Abschnitt 2.3.3 das mathematische Modell von aiT zur Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ genauer vorgestellt.

2.3.1 Analyse durch aiT

Die Analyse des Tools aiT basiert im Wesentlichen auf drei Bestandteilen: dem Programm in Form einer ausführbaren Datei, den zusätzlichen Informationen als Benutzerannotationen in der AIS-Datei sowie auf aiTs Modell der Zielarchitektur. Darauf basierend berechnet aiT in einer statischen Analyse die $\mathrm{WCET}_{\mathrm{est}}$ des Programms, liefert aber auch Zusatzinformationen über den Kontrollfluss bis hin zur Ausführungszeit einzelner Basisblöcke.

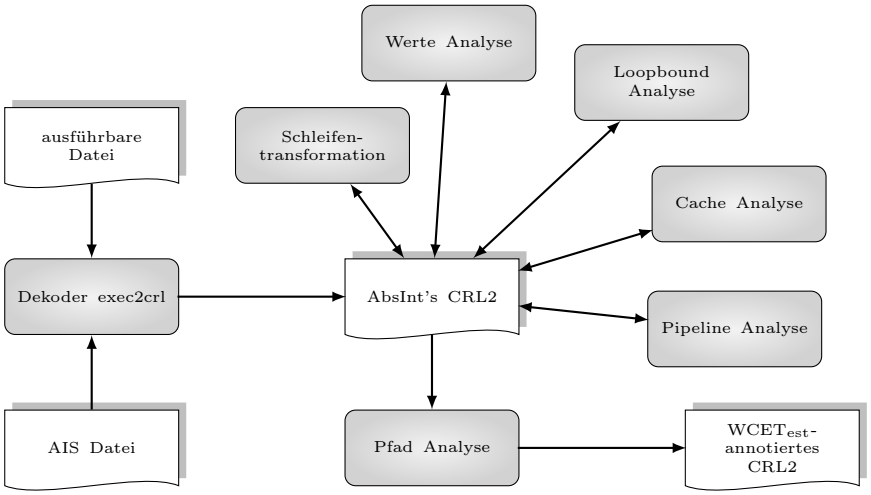


Abbildung 2.4: Aufbau von aiT

Die Analyse ist in mehrere Phasen untergliedert (vgl. Abb. 2.4):

- Der *Dekoder exec2crl* liest die ausführbare Datei sowie die AIS-Datei ein und rekonstruiert daraus den Kontrollflussgraph des Programms, annotiert mit den zusätzlichen Informationen des Anwenders. Dies wird in der internen Datenstruktur *CRL2* abgespeichert, die den folgenden Analysen als Grundlage dient und durch sie erweitert wird.
- In der *Schleifentransformation* werden Schleifen ermittelt und in einer eigenen Routine (CRL2s Gegenstück zur Funktion) gekapselt. Schleifen mit mehreren Einstiegspunkten (*Multi-Entry Loops*) werden i. d. R. nicht erkannt und dementsprechend auch nicht in eine eigene Routine gekapselt.
- In der *Werte Analyse* werden Wertebereiche für Registerinhalte sowie Adressbereiche für Speicherzugriffe ermittelt.
- Die *Loopbound Analyse* ermittelt für einfache Schleifen Grenzen für deren Iterationshäufigkeit.
- Die *Cache Analyse* klassifiziert Speicherzugriffe in *hits* und *misses*.
- Die *Pipeline Analyse* untersucht das Verhalten des Programms in der Pipeline des Zielprozessors.

- Die *Pfad Analyse* letztendlich bestimmt den längsten Pfad im Programm und damit die $WCET_{est}$.

Die Analysen beruhen i. d. R. auf *abstrakter Interpretation*, die Bestimmung des längsten Pfades in der Pfadanalyse allerdings auf *ILP* (*Integer Linear Programming*). Weitere Informationen zu aiT sind [AbsInt 2006] und [Heckmann u. Ferdinand 2004] zu entnehmen.

2.3.2 Annotationen für aiT

Das Tool aiT benötigt neben dem zu analysierenden Programm zusätzliche Informationen. Diese sind durch den Benutzer in einer eigenen Datei zu spezifizieren. Mögliche Informationen sind (Auswahl):

- Frequenz des Prozessors
- Ziele berechneter Sprünge
- Iterationshäufigkeit von Schleifen (*Loop Bounds*)
- Rekursionstiefe von Funktionen
- relative Ausführungshäufigkeit von Basisblöcken zueinander (*Flow Constraints*)
- ...

Nicht alle Informationen, die in der AIS-Datei spezifiziert werden können, sind Informationen zur eigentlichen Abschätzung der WCET. Es werden z. B. auch Informationen zur Rekonstruktion des Kontrollflusses wie Ziele berechneter Sprünge annotiert und das Analyseverhalten von aiT konfiguriert.

2.3.3 Berechnungsmodell von aiT

Die Berechnung der $WCET_{est}$ in der Pfadanalyse basiert auf einem um *Kontexte* erweiterten Kontrollflussgraphen. Dabei nutzt aiT das Konzept der Kontexte, um verschiedene Ausführungsbedingungen für einen Basisblock unterscheiden zu können und so eine genauere Analyse durchzuführen. Kontexte erlauben z. B. die Unterscheidung verschiedener Wertebereiche von Registerinhalten für die Ausführung eines Basisblocks. Je mehr Kontexte unterschieden werden, desto genauer ist potentiell das Ergebnis der Analyse.

Durch die Informationen der vorangegangenen Analyseschritte und letztendlich durch die Pipeline Analyse ist jede Kante des Kontrollflussgraphen e in Abhängigkeit eines Kontextes c mit einer lokalen $WCET_{est}$ $T(e, c)$ beschriftet worden. Mit den Entscheidungsvariablen $C(e, c)$, die jeder Kanten/Kontext Kombination ihre Ausführungshäufigkeit auf dem globalen $WCET_{est}$-Pfad zuordnen, ergibt sich folgendes Optimierungsproblem

$$\sum_{\forall\, e,c} C(e,c) \cdot T(e,c) \rightarrow max \tag{2.6}$$

unter einer Menge von Nebenbedingungen, die aiT aus dem Kontrollflussgraphen, aus den Analyseergebnissen und aus den Benutzerannotationen ermittelt.

Dabei bleibt anzumerken, dass die Benutzerannotationen nicht in Abhängigkeit von Kanten oder Kontexten formuliert werden, sondern sich stets auf Programmpunkte, also auf Knoten eines Kontrollflussgraphen, beziehen. Die Wandlung dieser Bezüge zu Entscheidungsvariablen des Optimierungsproblems wird intern durch aiT durchgeführt.

Wird im Folgenden im Zusammenhang mit Annotationen von Referenzen auf eine Anweisung, ein Statement oder einen Basisblock gesprochen, so ist der Bezug auf die Entscheidungsvariable gemeint, die diesem Element seine Ausführungshäufigkeit auf dem WCET-Pfad zuordnet.

2.4 Informationen zur WCET-Approximation

In Anbetracht der Unentscheidbarkeit der $\mathrm{WCET}_{\mathrm{real}}$-Berechnung sind Informationen von besonderem Interesse, die

1. eine Abschätzung der $\mathrm{WCET}_{\mathrm{real}}$ ermöglichen und
2. die Qualität der Abschätzung steigern.

Denn obwohl Analysetools wie aiT automatisch versuchen, alle notwendigen Informationen zu ermitteln – z. B. die Iterationshäufigkeit von Schleifen – gelingt dies nur in einigen Fällen. Es verbleibt für die WCET-Analyse eine **Informationslücke**, die durch den Anwender zu schließen ist.

Folgendes Beispiel zeige die Grenzen der automatischen Analyse durch aiT:

Beispiel: *Für das Programm* bubblesort *in Abb. 2.5 auf der nächsten Seite wird die Ausführungshäufigkeit der verschachtelten Schleife in den Zeilen 10 und 11 von aiT mit insgesamt 45 Ausführungen des inneren Schleifenkörpers pro Funktionsaufruf präzise bestimmt. Dagegen gelang es aiT jedoch nicht, eine maximale Iterationshäufigkeit für die Schleife in Zeile 22 zu ermitteln. Dementsprechend ist die Abschätzung der* $WCET_{real}$ *erst möglich, wenn diese Schleife durch eine Annotation des Benutzers beschränkt wird.*

Annotationen für $\mathsf{WCET}_{\mathsf{est}}$

Allgemein sind für folgende Elemente Annotationen zur Berechenbarkeit der $\mathrm{WCET}_{\mathrm{est}}$ notwendig [vgl.Vrchoticky 1993; Kirner 2002; Kirner u. Puschner 2005a; Heckmann u. Ferdinand 2004]:

- **Schleifen**
  Wie im Beispiel zuvor gesehen, muss die Iterationshäufigkeit einer Schleife i. d. R. vom Benutzer begrenzt werden.
- **(direkte und indirekte) Rekursion**
  Die Anzahl der rekursiven Aufrufe einer Funktion muss beschränkt werden. Alternativ verbieten viele Ansätze zur WCET-Analyse Rekursionen vollständig [z. B. Vrchoticky 1993; Kirner 2002].

```
                                    /* bubblesort für Integerarray: */
2       /* Größe frei wählbar, aber Maximum MAX_N notwendig, da sonst WCET_real unbeschränkt. */

4 #define MAX_N 10
  int values[MAX_N];
6
  void bubble( int n )
8 {
    int hilf, i, j;
10  for ( i = n - 1; i > 0; --i )
      for ( j = 0; j < i; ++j )
12      if ( values[j] > values[j+1] ){
          hilf = values[j];
14        values[j] = values[j+1];
          values[j+1] = hilf;
16      }
  }
18
  int main( void )
20 {
    int i;
22  for ( i = 0; i < MAX_N; ++i )
      values[i] = MAX_N - i;
24  bubble( MAX_N );
    return 0;
26 }
```

Abbildung 2.5: Bsp. für die Notwendigkeit von Annotationen

- **implizite Schleifen**
  Durch Sprunganweisungen wie die *goto*-Anweisung der Programmiersprache C können implizit Schleifen programmiert werden, d. h. Schleifen ohne Nutzung der dafür vorgesehenen Sprachkonstrukte wie *for*-, *while*- oder *do-while*-Anweisungen. Auch deren Iterationshäufigkeit ist vom Benutzer zu begrenzen. Wiederum wird die Nutzung dieser Konstrukte in einigen Ansätzen verboten [z. B. Vrchoticky 1993; Kirner 2002].

Zusammenfassend sind all jene Sprachkonstrukte mit Annotationen zu ihrer maximalen Iterationshäufigkeit zu versehen, die im Kontrollflussgraphen zu Kreisen führen.

Definition 2.6
Annotationen, die die Iterationshäufigkeit von Kreisen im globalen Kontrollflussgraphen beschränken, werden als **grundlegende Begrenzungsinformationen** bezeichnet (basic finiteness informations) [nach Engblom u. Ermedahl 2000].

Weitere Kontrollflussinformationen wie z. B.

- die relative Ausführungshäufigkeit von Anweisungen zueinander,
- die absolute Ausführungshäufigkeit von Anweisungen,
- die Exklusivität der Ausführung von Anweisungen,

- ...,

also all jene Informationen, die ebenfalls unausführbare Pfade spezifizieren, können die Qualität der $\mathrm{WCET}_{\mathrm{est}}$ verbessern.

3 WCET-optimierende Compiler

Während anfangs die Programmierung Eingebetteter Systeme direkt in Assembler stattfand, werden wegen der gewachsenen Komplexität derer Aufgaben und damit auch derer Programmierung heute Hochsprachen wie C dafür eingesetzt. *Compiler* erzeugen aus dem Quellcode eines Programms dessen (optimierten) Assemblercode. Dieser ist Ausgangspunkt für gängige WCET-Analysen, die nach der Übersetzung des Programms manuell zu starten sind.

Um die Ergebnisse der WCET-Analyse automatisch verfügbar zu machen und so optimal in den Softwareentwicklungsprozess einzubinden, muss die Analyse in den Compiler integriert werden. Dies bietet zugleich die Chance, deren Ergebnisse für Optimierungen zu nutzen und somit die (berechnete) WCET zu reduzieren.

Ein entsprechender Compiler namens *WCC* (*WCET-Aware C Compiler*) für den C99-Standard [ISO 9899 1999] wird am Lehrstuhl 12 der Universität Dortmund entwickelt [Falk u. a. 2006] und in Abschnitt 3.1 vorgestellt.

Auf dieser Basis wird die Problemstellung für die Diplomarbeit in Abschnitt 3.2 erarbeitet und eine Lösungsstrategie zum Erreichen derer Ziele in Abschnitt 3.3 präsentiert.

3.1 WCC

Der Aufbau des WCCs folgt dem eines klassischen Compilers wie z. B. in [Bacon u. a. 1994] beschrieben. Er wird in Abb. 3.1 auf der nächsten Seite dargestellt.

Ein C99 Parser übersetzt den ANSI-C Code in **ICD-C**, die Intermediate Representation des C nahen *Frontend*. Die Intermediate Representation **ICD-LLIR** des *Backend* dagegen ist Assembler nah. Die Übersetzung von Frontend nach Backend erfolgt durch den **LLIR Code Selector**. Der Code Generator erzeugt letztendlich aus dem Backend den reinen Assemblercode.

ICD-C wird in Abschnitt 3.1.1, die ICD-LLIR in Abschnitt 3.1.2 und der LLIR Code Selector in Abschnitt 3.1.3 detailliert vorgestellt. Die Einbindung einer WCET-Analyse durch aiT in diesen Compiler wird im Abschnitt 3.1.4 dargestellt.

Zielarchitektur für den WCC im Rahmen der Diplomarbeit ist der TriCore Prozessor von Infineon [Infineon 2005a, b]. Die TriCore Architektur ist eine 32-bit single-core DSP-Architektur, optimiert für den Einsatz in Eingebetteten Systemen.

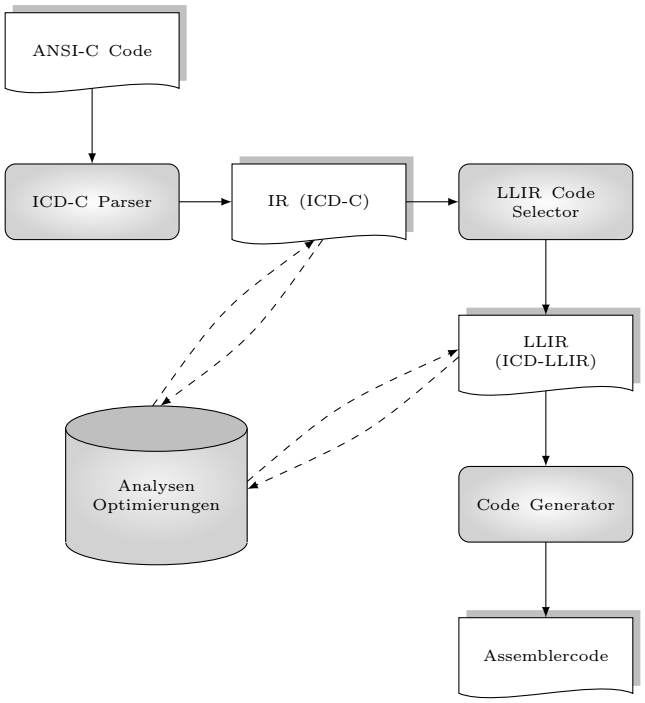


Abbildung 3.1: WCC: Compileraufbau

3.1.1 ICD-C

Die High Level Intermediate Representation ICD-C (kurz IR) des WCCs wurde vom Informatik Centrum Dortmund [ICD 2007] entwickelt. Sie enthält alle Sprachkonstrukte wie Schleifen (*for*, *while*, *do-while*) oder Verzweigungen (*if*, *if-else*, *switch*) der Programmiersprache C und ist unabhängig von einer Zielplattform. Ihr Aufbau wird in Abb. 3.2 auf der nächsten Seite skizziert. Eine detaillierte Beschreibung ist in [ICD-C 2006] zu finden.

Aufbau

Die IR als zentrale Instanz verwaltet alle Compilation Units (IR_CompilationUnit) eines Projekts (Teile eines Programms mit eigenem Namensraum, die separat durch einen Compiler übersetzt werden können) sowie deren globale Symbole. Die Compilation Units bestehen aus Funktionen (IR_Function). Jede Funktion wird durch Statements (IR_Stmt) realisiert, die in einem ausgezeichneten Compound Statement (IR_CompoundStmt), dem Top Compound Statement, in der Reihenfolge ihrer Abarbeitung verwaltet werden. Dabei bleiben die durch die Semantik eines Statements bedingten Änderungen der sequentiellen Abarbeitungsreihenfolge wie z. B. bei Sprüngen oder Schleifen unberücksichtigt.

Entsprechend des C Sprachumfangs bietet ICD-C folgende Statements: Expression Statements (IR_ExpStmt), Jump Statements (IR_JumpStmt), Loop Statements (IR_LoopStmt), Selection Statements (IR_SelectionStmt), Targeted Statements (IR_TargetedStmt) sowie Compound

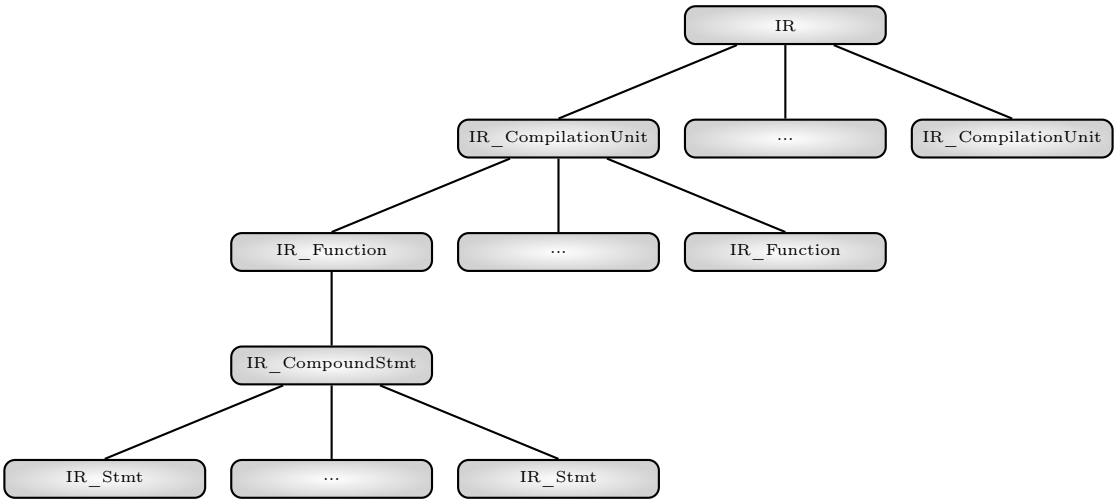


Abbildung 3.2: ICD-C Aufbau (abstrahiert)

Statements. Zum Teil werden diese Statements noch weiter unterteilt wie Abb. 3.3 auf der nächsten Seite zeigt.

Die Statements sind hierarchisch geordnet. So haben Compound, Loop und Selection Statements Nachfahren.

- Compound Statements fassen beliebig viele Statements zu einem Anweisungsblock zusammen.
- Loop Statements haben einen Schleifenkörper (*Loopbody*), als Compound Statement realisiert. Darüber hinaus verfügen *for*-Statements über ein Init Clause Statement – ein spezielles Compound Statement mit ggf. einem Expression Statement – für ihren Deklarationspart, ggf. über ein Expression Statement als Abbruchbedingung und ggf. über ein Expression Statement als Folgeausdruck (das Statement wird nach jeder Ausführung des Schleifenkörpers ausgeführt).
- Der *then*-Anweisungsblock eines *if*- oder *if-else*-Statements ist als Compound Statement realisiert, ebenso wie der *else*-Anweisungsblock des *if-else*-Statements und der *select*-Anweisungsblock des *switch*-Statements.

Als Abbruchbedingung in einem *do-while*- oder *while*-Statement sowie als Teil eines Expression Statements oder eines *return*-Statements gibt es verschiedene Expressions (IR_Exp). Dies sind Ausdrücke wie z. B. Zuweisungen oder binäre Operationen.

Pragmas

Pragma Direktiven können im C Code als Präprozessordirektive (#pragma) oder per Pragma-Operator (_Pragma) annotiert werden. Sie werden durch den ICD-C Parser dem nächsten folgenden Element der ICD-C angehängt, das eine Symboldeklaration, ein Statement, eine Expression oder ein Typ (dient der Beschreibung der verschiedenen Datentypen) ist.

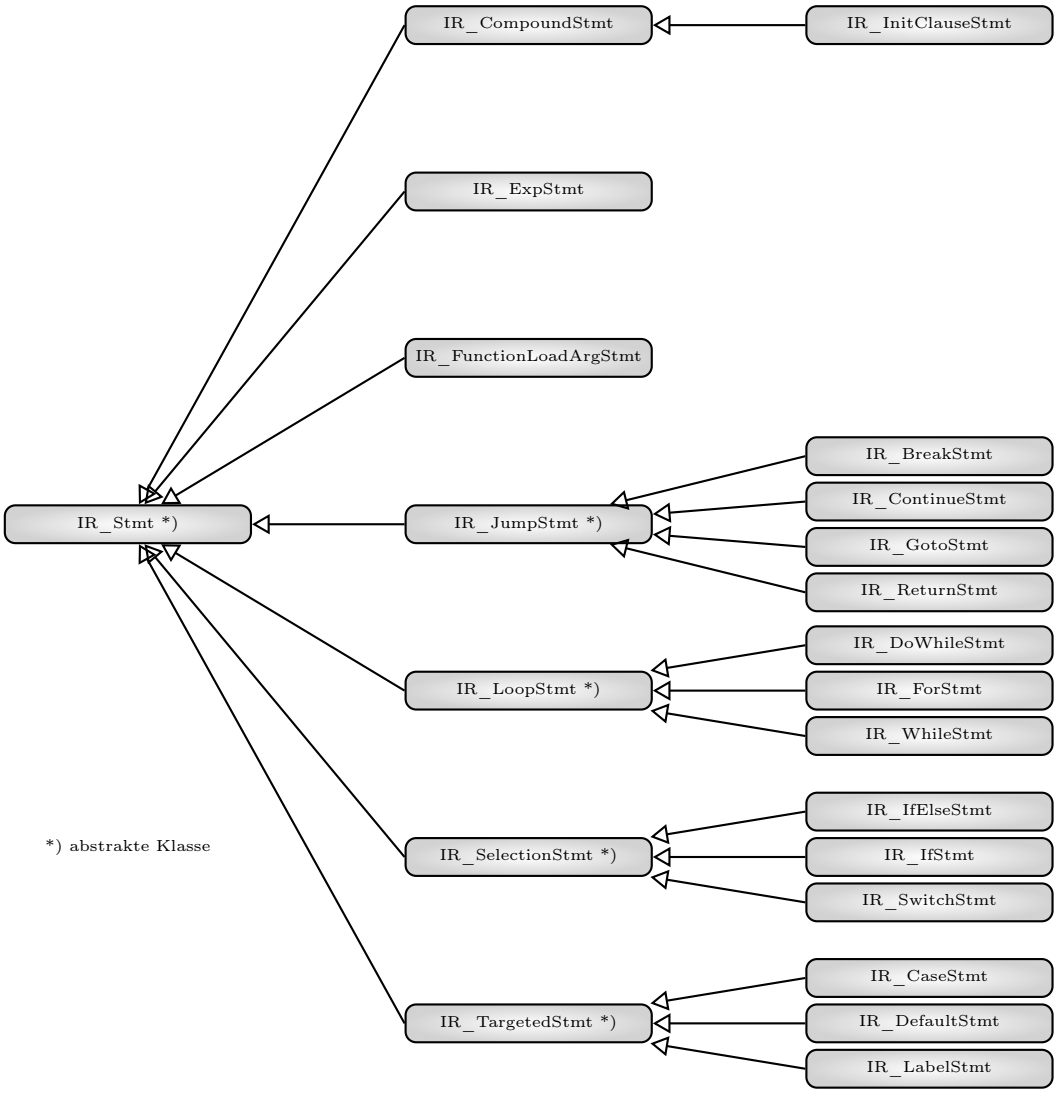


Abbildung 3.3: ICD-C Statements

Analysen

ICD-C bietet diverse Möglichkeiten, um die Datenstruktur zu analysieren, so z. B. durch Kontrollfluss- oder Datenflussanalysen. Vor allem die Möglichkeit, mit einer *Callback Function* durch die Statement oder Expression Hierarchie zu iterieren, ermöglicht komfortabel, eigene Analysen zu ergänzen. Darüber hinaus werden Manipulationen an allen Elementen der Datenstruktur unterstützt.

ICD-Cs Kontrollflussanalyse ermöglicht u. a. die Ermittlung von Basisblöcken für eine Funktion. Diese fassen je eine Reihe von Statements zusammen und sind über die entsprechende Funktion und von den betroffenen Statements aus erreichbar. Zudem werden für jeden Basisblock Vorgänger und Nachfolger verwaltet. Die Analyseergebnisse gelten jedoch nur bis zur nächsten, potentiell den Kontrollflussgraph modifizierenden Änderung der ICD-C Datenstruk-

tur und müssen danach erneut berechnet werden.

Schleifen können durch einen Loop Analyzer untersucht werden, inwiefern sie sich z. B. für Optimierungen eignen. Der Loop Analyzer kann u. U. für einfache Schleifen gar eine genaue Iterationshäufigkeit ermitteln.

Benutzerdaten

Die ICD-C Datenstruktur kann um Benutzerdaten – hier *User Data* genannt – erweitert werden. Dazu wurde jede Klasse der Datenstruktur von der Klasse IR_PersistentObject abgeleitet. Diese bietet insbesondere die Möglichkeit der Speicherung der gesamten Datenstruktur. Aber sie stellt auch die Funktionen

- setUserData(ITEM_t key, IR_PersistentObject *data);
- getUserData(ITEM_t key); und
- removeUserData(ITEM_t key);

zur Verfügung, durch die ein beliebiges Objekt einer ebenfalls von IR_PersistentObject abgeleiteten Benutzerklassen an jedes Objekt der Datenstruktur angehangen werden kann. Dabei werden diese angehängten Objekte über einen Namen key eindeutig identifiziert.

Vorteil dieses Verfahrens ist, dass die Datenstruktur um beliebige Daten erweitert werden kann, ohne die ICD-C selbst zu verändern.

Optimierungstechniken

Die ICD-C bietet plattformunabhängige Quellcode Optimierungen: Standard Optimierungen wie *Dead Code Elimination* oder *Value Propagation*, aber auch Optimierungen von Funktionsaufrufen wie *Function Specialization* oder Optimierungen von Schleifen wie *Loop Unrolling*. Eine Übersicht aller Optimierungstechniken ist im Anhang A.1 auf Seite 113 zu finden.

3.1.2 ICD-LLIR

Die Low Level Intermediate Representation ICD-LLIR (kurz LLIR) des WCCs stammt ebenfalls vom Informatik Centrum Dortmund. Im Gegensatz zu ICD-C ist sie jedoch Assembler nah, d. h. in der Datenstruktur werden u. a. die Mnemonics der Assemblerbefehle verwaltet. Dementsprechend ist ihre Darstellung abhängig von der Zielplattform. Dies ermöglicht plattformspezifische Optimierungen. Der Aufbau wird in Abb. 3.4 auf der nächsten Seite skizziert. Eine detaillierte Beschreibung ist in [ICD-LLIR 2006] zu finden.

Aufbau

Die LLIR repräsentiert als zentrale Klasse ein ganzes Programm oder den Teil eines Programms. Sie entspricht im WCC einer Compilation Unit der IR. Die LLIR hält Funktionen (LLIR_Function), die sich wiederum aus Basisblöcken (LLIR_BB) zusammensetzen. Für jeden Basisblock werden die Vorgänger und Nachfolger im Kontrollfluss verwaltet. Alle Basisblöcke

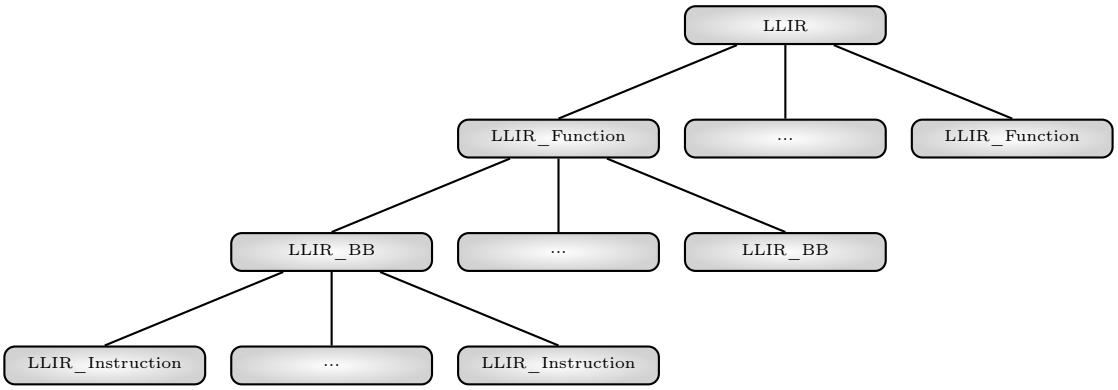


Abbildung 3.4: ICD-LLIR Aufbau (abstrahiert)

werden durch ein eindeutiges Label identifiziert. Der sequenzielle Code innerhalb der Basisblöcke wird durch Instruktionen (LLIR_Instruction) modelliert. Der genaue Aufbau der Instruktionen sowie die Verwaltung von Registern ist im Rahmen dieser Arbeit unbedeutend.

Analysen

Die LLIR bietet nicht nur Kontrollflussanalysen, sondern z. B. auch die Möglichkeit, einen Kontrollflussgraphen auszugeben. Ferner sind Analysen des Datenflusses wie z. B. eine *Live Time Analysis* Bestandteil der LLIR. Darüber hinaus werden wie in ICD-C Techniken zur Manipulation aller Elemente der Datenstruktur zur Verfügung gestellt.

Benutzerdaten

Auch die ICD-LLIR Datenstruktur kann um Benutzerdaten erweitert werden. Die LLIR Elemente bieten dazu durch die Klasse LLIR_Handler über die Funktionen

- addObjective(LLIR_Objective *obj);
- getObjective(ObjectiveType type); und
- hasObjective(ObjectiveType type);

die Möglichkeit, *Objectives* zu verwalten. Objectives sind dabei Objekte einer beliebigen Benutzerklasse, die durch Ableitung der abstrakten Klasse LLIR_Objective erzeugt und eindeutig über ihren ObjectiveType identifiziert wird.

Vorteil dieses Verfahrens ist wieder, die Datenstruktur um beliebige Benutzerdaten erweitern zu können, ohne die ICD-LLIR selbst zu verändern.

Optimierungstechniken

Die ICD-LLIR bietet verschiedene Optimierungen. Neben Techniken zur

- Entfernung leerer Basisblöcke und zur
- Entfernung unbenutzter, virtueller Register

sind aber insbesondere plattformabhängige Optimierungen möglich. Für diese werden verschiedene Basisklassen zur Verfügung gestellt, die durch Überschreiben von Funktionen die individuelle Anpassung der Optimierung für eine Zielplattform erlauben. Eine Übersicht aller Optimierungen ist im Anhang A.2 auf Seite 117 zu finden.

3.1.3 LLIR Code Selector

Aufgabe des LLIR Code Selectors ist die Übersetzung einer (optimierten) ICD-C Datenstruktur in eine ICD-LLIR Datenstruktur. Dazu wird für jede Compilation Unit inkl. Funktionen eine eigene LLIR mit denselben Funktionen erzeugt. Für jeden ICD-C Basisblock einer Funktion wird schließlich ein von ICD-C bereitgestelltes *Tree Pattern Matching* durchgeführt und der entstandene Baum zur Auswahl von Assembler Instruktionen für die Zielarchitektur genutzt.

Die notwendigen Regeln zur Überdeckung des Baumes inkl. einer Kostenfunktion zur Auswahl der günstigsten Überdeckung werden für den *Code-Generator Generator* **Olive++** [ICD-CG 2007] spezifiziert. Wie in [Muchnick 1997; Vasseur 2004] für ähnliche Code-Generator Generatoren beschrieben, erzeugt auch Olive++ aus dem Regelwerk schließlich einen *Code Selector*. Dieser nutzt die Technik der dynamischen Programmierung, um kostengünstige Überdeckungen eines Baumes in linearer Zeit zu finden.

Für diese Arbeit von besonderem Interesse ist, dass durch den Code Selector Schleifen erzeugt werden können, wie z. B. für die richtige Initialisierung von zusammengesetzten Datentypen (*structs*, *unions*) sowie Arrays.

3.1.4 Einbindung der WCET-Analyse

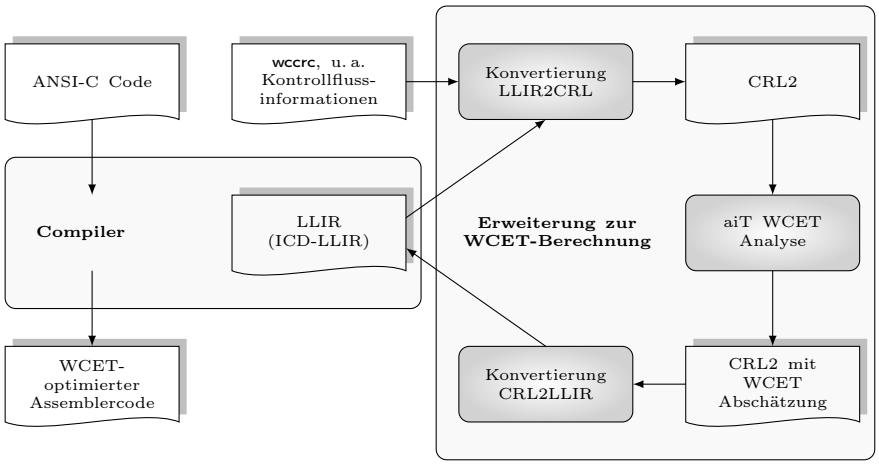


Abbildung 3.5: WCC: Einbindung aiT

Eine Besonderheit des WCCs ist seine integrierte WCET-Analyse durch aiT (Abb. 3.5). Für deren Einbindung wird allerdings nicht aiTs Interface per AIS- und Binär-Datei genutzt. Statt

dessen wird die LLIR direkt durch die **Konvertierung LLIR2CRL** in das aiT interne Datenformat *CRL2* (*Control flow Representation Language*) übersetzt. Damit ist aiT – dann ohne seinen Dekoder exec2crl – aufzurufen.

Dies bietet den Vorteil, dass z. B. Kontrollflussinformationen aus der LLIR nach CRL2 übernommen werden können, aber auch, dass die Ergebnisse der WCET-Analyse durch aiT wegen einer eindeutigen Abbildung der LLIR-Elemente auf CRL2-Elemente später zurück gelesen werden können. Die Einbindung ist in [Lokuciejewski 2005] ausführlich dargestellt.

Kontrollflussinformationen

Informationen zum möglichen Kontrollfluss des zu analysierenden Programms können nicht mehr in der AIS-Datei (deren Auswertung erfolgt in der entfallenen exec2crl Dekodierung) sondern nur noch in der WCC eigenen Spezifikationsdatei wccrc gegeben werden. Im Moment sind Informationen über Schleifen (Minimum und Maximum der Iterationshäufigkeit) sowie über Rekursionen (Aufruftiefe und -häufigkeit) annotierbar.

Die Annotation von z. B. Schleifen erfolgen analog zu deren Annotation in der AIS-Datei per:

LOCAL : Routine name, Number of loop, min, max

Die Schleife wird also über ihren Routinenamen sowie ihrer Nummer nach Durchnummerierung aller Schleifen der entsprechenden Routine identifiziert. Bei einer Rekursion wird nur der Routinename benötigt.

Zu beachten ist, dass die Annotationen (sowohl der Wert, also z. B. min und max, als auch die Identifizierung) für die Programmstruktur nach Übersetzungen und möglichen Optimierungen durch den Compiler gelten müssen. Sie sind also nicht für die Programmstruktur im Quellcode, sondern für dessen Darstellung in der LLIR anzugeben. Das heißt aber auch, dass je Optimierungsstufe (der WCC unterstützt O0 bis O3) spezifische Annotationen notwendig sind.

3.2 Problemstellung

Im Abschnitt 2.4 auf Seite 12 wurden Informationen zur Approximation der WCET vorgestellt, um die Informationslücke, die durch die Unentscheidbarkeit der $\mathrm{WCET}_{\mathrm{real}}$-Berechnung entsteht, zu schließen.

Wie in Abb. 3.5 auf der vorherigen Seite zu sehen ist, werden diese Informationen im WCC z. Z. durch den Einsatz von Annotationen in der wccrc-Datei gegeben. Dabei müssen sich diese Annotationen auf die Struktur des Programms in der LLIR bzw. in CRL2 beziehen. Der Kontrollfluss eines Quellprogramms wird jedoch vor allem durch Optimierungen, aber auch durch die Code Selection wesentlich modifiziert. Dabei hängen Art und Umfang der Modifikationen vom Programm selbst aber auch von der gewählten Optimierungsstufe ab.

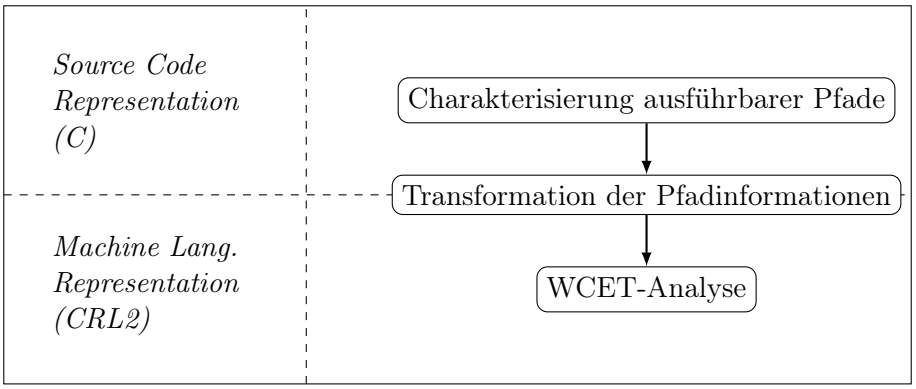
Für den Programmierer bedeutet dies, nicht nur für den Quellcode entsprechende Annotationen zu ermitteln, sondern diese zusätzlich unter hohem Aufwand zu transformieren, um sie kompatibel zur LLIR bzw. zu CRL2 zu annotieren. Dafür sind detaillierte Kenntnisse des

Compilers und seiner Optimierungen notwendig. Für jede Optimierungsstufe ist eine eigene Transformation der für den Quellcode gefundenen Annotationen notwendig. Zudem kann jede kleine Änderung im Quellcode – auch ohne eine Benutzerannotation direkt zu betreffen – Auswirkungen auf die Transformationen eines Programms durch den Compiler haben und ist daher auf mögliche Einflüsse für die Transformation von Annotationen zu untersuchen.

Aufwand und Fehleranfälligkeit dieser Annotationen und Transformationen von Hand sind nicht akzeptabel. Daher ist Ziel der Diplomarbeit, diese zusätzlich benötigten Kontrollflussinformationen im Quellcode eines Programms zu annotieren, und für eine Verarbeitung durch aiT automatisch aufzubereiten (vgl. Abschnitt 1.2).

3.3 Lösungsansatz

Das Problem der WCET-Analyse wird in [Puschner u. Burns 2000] in folgende drei Bereiche gegliedert (vgl. Abb. 3.6):



Graphik nach [Puschner u. Burns 2000]

Abbildung 3.6: Probleme der WCET-Analyse

1. Die ausführbaren Pfade eines Programms müssen auf Quellcode Ebene beschrieben werden können, da sowohl eine manuelle als auch eine automatische Generierung dieser Informationen auf niedrigeren Darstellungsebenen ungleich schwerer ist.
2. Die Transformation dieser Informationen von der Quellcode Ebene zur Assembler Ebene (hier Assembler nahes CRL2) – vor allem wegen der Codeumstellungen durch Optimierungstechniken komplex – ist zu automatisieren.
3. Die Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ für ein Programm(teil) ist schließlich hardwarenah vorzunehmen.

Analog zu diesen drei Schritten werden die Ziele der Diplomarbeit erreicht. Im Detail bedeutet dies:

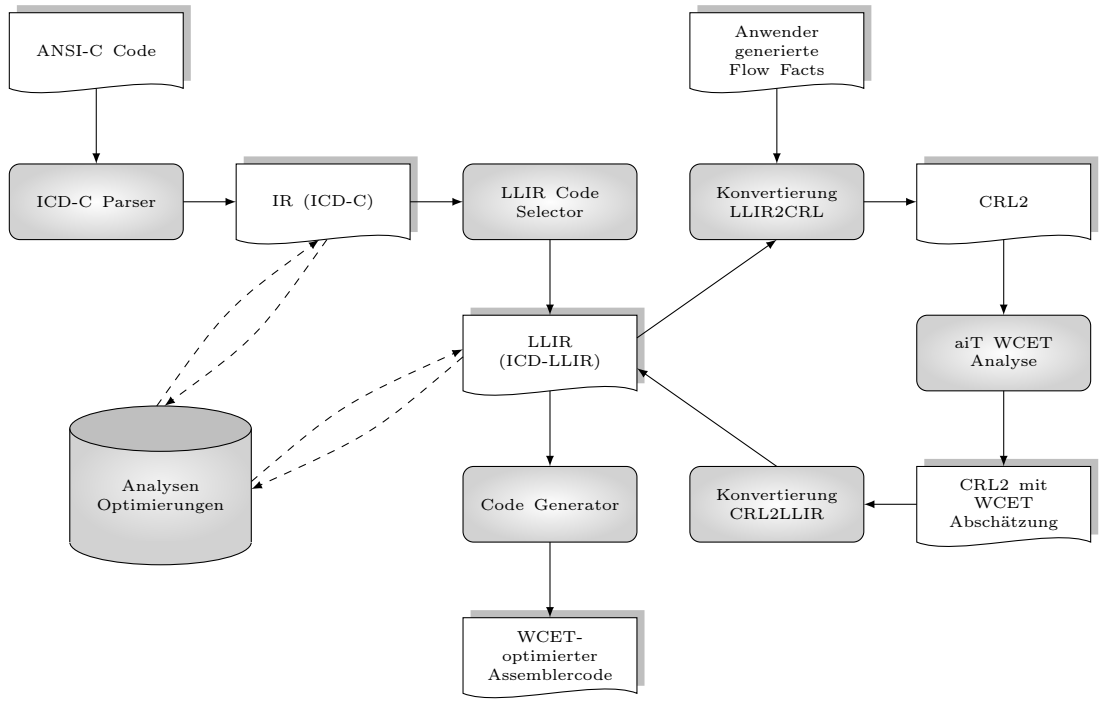


Abbildung 3.7: WCC: Aufbau vor der Diplomarbeit

1. **Modellierung von Flow Facts** (Kapitel 4)
   Einer Analyse, welche Arten von Annotationen notwendig sind, folgt die Modellierung derer Struktur, zum einen als Datenstruktur im WCC, zum anderen als Schnittstelle für den Benutzer im Quellcode.
2. **Transformation von Flow Facts** (Kapitel 5)
   Die im Quellcode beschriebenen Annotationen werden im assemblernahen CRL2 benötigt. Bis dahin ist das Programm diversen Transformationen ausgesetzt:
   a) Flow Fact Extraktion, Code Selection und Konvertierung ins CRL2
   b) Optimierungen auf der IR und der LLIR

   Die Annotationen sind ebenfalls entsprechend zu transformieren.
3. **Berechnung der $\mathbf{WCET_{est}}$**
   Die Berechnung der $\mathrm{WCET_{est}}$ erfolgt wie schon dargestellt durch aiT und ist nicht Gegenstand dieser Arbeit.

Durch diese Schritte entfällt die Annotierung von Flow Facts in der wccrc-Datei, die aber weiterhin für statische Informationen wie z. B. Adressbereiche genutzt werden wird. Ebenso entfällt die manuelle Transformation von Flow Facts.

Die Abstraktionsebenen ANSI-C Code, ICD-C und ICD-LLIR werden erweitert, um Flow Facts zu halten. Der LLIR Code Selector und die Konvertierung LLIR2CRL werden modifiziert, um die Flow Facts zwischen den einzelnen Abstraktionsebenen zu übersetzten, die Flow Fact Extraktion gewinnt diese aus dem Quellcode eines Programms.

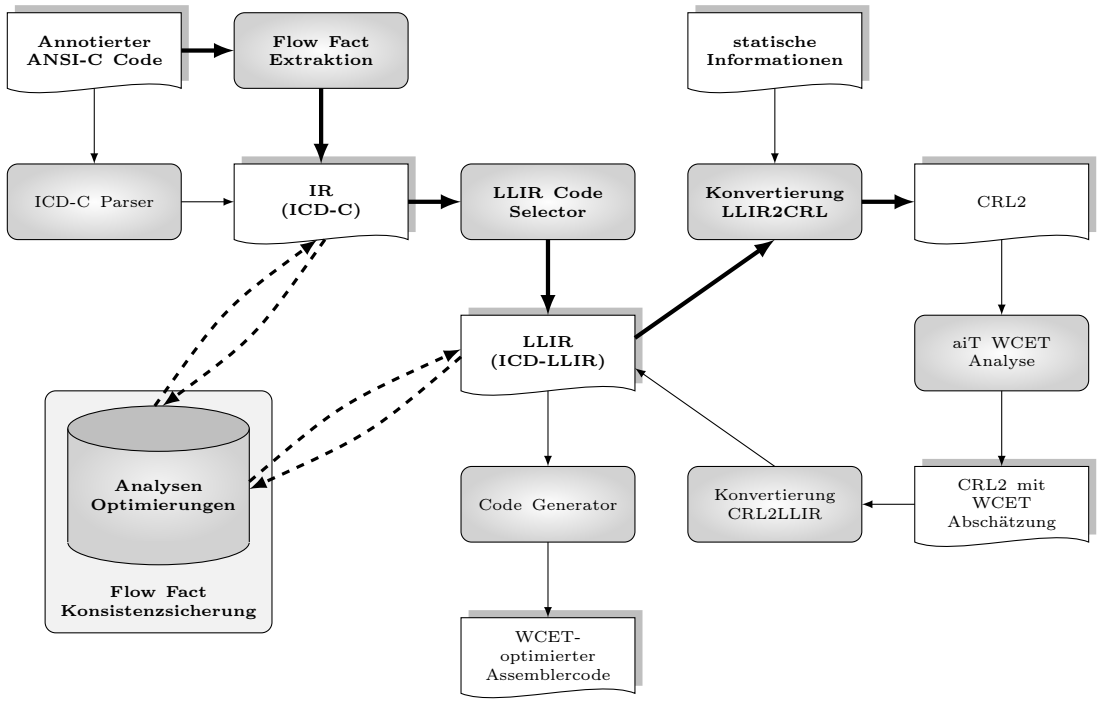


Abbildung 3.8: WCC: Änderungen durch die Diplomarbeit

Für die Optimierungen von ICD-C und ICD-LLIR werden Konsistenzsicherungsmechanismen ergänzt.

Der Aufbau des WCCs vor und nach der Diplomarbeit wird in den Abb. 3.7 auf der vorherigen Seite und 3.8 gegenübergestellt. Die von Veränderungen betroffenen Elemente sind zudem in der zweiten Abbildung durch Fettdruck hervorgehoben.

4 Modellierung von Flow Facts

Die Modellierung von Flow Facts für den WCET-optimierenden Compiler WCC ist Ziel dieses Kapitels.

Die Einführung einer Annotationsklassifikation in Abschnitt 4.1 bietet einen ersten Überblick über mögliche Annotationen und eine Bewertung dieser für deren Einsatz im WCC. In Abschnitt 4.2 wird der Begriff Flow Facts für einen Teil dieser Annotationen eingeführt. Abschnitt 4.3 gewährt einen Überblick über Modellierungen von Flow Facts in vorausgegangenen Arbeiten zu dieser Thematik.

Das Design der Flow Facts wird schließlich in den drei Schritten Anforderungsanalyse in Abschnitt 4.4, Designentwicklung und Beschreibung in Abschnitt 4.5 sowie Implementierung in Abschnitt 4.6 behandelt.

4.1 Annotationen zur WCET_{est}-Berechnung

Zur Berechnung der WCET_{est} mit aiT sind neben dem zu analysierenden Programm weitere Informationen zu spezifizieren. In [Kirner u. Puschner 2005a] werden diese Informationen in vier Kategorien untergliedert:

- **Annotationen zu Plattformeigenschaften** (*Platform Property Annotations*)
  Diese Annotationen sind statische Informationen z. B. über die aktuelle Konfiguration der Hardware. Sie werden derzeit in der Konfigurationsdatei wccrc spezifiziert. Da diese Informationen weder vom Programmcode abhängig sind, noch im WCC transformiert werden müssen, ist eine Konfigurationsdatei ideal dafür geeignet.
- **Annotationen zur Kontrollflussrekonstruktion** (*Control Flow Graph Reconstruction Annotations*)
  Da aiT keinen Assemblercode einlesen muss, um aus diesem den Kontrollfluss zu rekonstruieren, sondern eine direkte Übersetzung der LLIR in aiTs interne Intermediate Representation CRL2 erfolgt, sind keine Annotationen zur Kontrollflussrekonstruktion im WCC notwendig.
- **Annotationen zur Programmsemantik** (*Program Semantics Annotations*)
  Diese Annotationen sind Informationen, wie sie in Abschnitt 2.4 auf Seite 12 beschrieben werden. Sie sind sowohl vom Programm als auch von den Transformationen im Compiler abhängig. Zur Zeit werden sie mit Bezug auf den Assemblercode eines Programms in der Konfigurationsdatei wccrc annotiert. Ziel ist eine Annotation dieser Informationen mit Bezug auf den Quellcode eines Programms.

- **Hilfsannotationen** (*Auxiliary Annotations*)
  Diese Annotationen haben selbst keine Funktion für die WCET-Analyse, helfen aber, z. B. bestimmte Codeabschnitte zu referenzieren, so dass auf diese für andere Annotationen effizient und komfortabel zurückgegriffen werden kann.

Die Anmerkungen zu den Kategorien zeigen, dass insbesondere die Annotationen zur Programmsemantik von besonderem Interesse sein werden. Diese werden auch als Flow Facts bezeichnet.

4.2 Definition von Flow Facts

Der Begriff *Flow Facts* wird von Raimund Kirner in seiner Dissertation [Kirner 2003] geprägt und lässt sich wie folgt definieren:

Definition 4.1
Flow Facts sind Metainformationen, die Hinweise über die mögliche Menge aller Kontrollflusspfade eines Programms $\mathcal{P}$ geben. Die resultierende Kontrollflusspfadmenge wird mit $CFP_{ff}(\mathcal{P})$ bezeichnet.

Zwei Klassen von Flow Facts können unterschieden werden. Flow Facts können zum Einen durch Struktur und Semantik eines Programms gegeben sein, zum Anderen können sie explizit durch den Anwender vorgegeben werden.

Definition 4.2
Flow Facts, die durch die Struktur und Semantik eines Programms gegeben sind, heißen **implizite Flow Facts** (ff_{impl}), jene explizit gegebenen dagegen **annotierte Flow Facts** (ff_a).

Die Annotation von Flow Facts ist insbesondere dann notwendig, wenn der Kontrollfluss von zur Compilezeit unbekannten Parametern abhängt, aber auch, wenn implizite Flow Facts bei der WCET-Analyse nicht automatisch durch das entsprechende Tool ermittelt werden können. Jene impliziten Flow Facts sind dann explizit zu annotieren, um sie für die WCET-Analyse verfügbar zu machen. Dementsprechend müssen die Flow Fact Mengen ff_{impl} und ff_a nicht disjunkt sein.

Im Folgenden sind unter dem Begriff Flow Facts i. d. R. annotierte Flow Facts zu verstehen, da deren Modellierung und Transformation Ziel dieser Arbeit sind.

4.3 Flow Facts in der Literatur

Da Flow Facts wegen der Unentscheidbarkeit der WCET-Analyse für diese zwingend notwendig sind, gibt es verschiedene Ansätze zur Modellierung von Flow Facts in der Literatur. Eine Auswahl wird im Folgenden untersucht.

- **aiT**
  Das Tool aiT wurde bereits in Abschnitt 2.3 auf Seite 9 vorgestellt. Flow Facts werden dort in einer separaten Datei spezifiziert.

- **Bound-T**
  Das WCET-Analyse-Tool Bound-T [Holsti u. Saarinen 2002; Bound-T 2005] ist ähnlich angelegt wie aiT und ermittelt u. a. für einfache Zählschleifen Iterationshäufigkeiten, benötigt aber z. B. annotierte Flow Facts (hier *Assertions* genannt) für komplexe Schleifen. Diese werden analog zu denen von aiT spezifiziert.

- **wcetC**
  Für die WCET-Analyse wird in wcetC [Kirner 2000, 2002] die Programmiersprache C erweitert, um entsprechende *Laufzeitinformationen* zu annotieren. So sind z. B. Schleifen wie folgt mit einer maximalen Iterationshäufigkeit zu versehen.

  for(i = 0, i < 50, i++) maximum 50 iterations {...}

  Marker sind Identifier und erlauben, im Quellcode des Programms Pfade zu benennen. Mit ihnen kann der mögliche Kontrollfluss in einem *Scope* (Blockelement ähnlich dem Compound Statement) mittels Ungleichungen im Verhältnis zur Ausführungshäufigkeit des Startknotens der Scope beschrieben werden. Schleifenbegrenzungen werden ebenfalls in diese Struktur umgewandelt.

- **Matlab/Simulink**
  In [Kirner u. a. 2002] wird ein Ansatz ohne Benutzerannotationen vorgestellt. Der Entwurf des Systems wird in Matlab/Simulink vorgenommen. Hier sind scheinbar alle notwendigen Flow Facts implizit vorhanden, so dass sich aus diesem Entwurf (annotierter) wcetC Code erzeugen lässt. Auf diesem basiert schließlich die WCET-Analyse.

- **Flow Information**
  In [Engblom u. Ermedahl 2000] wird eine Repräsentation von Flow Facts (hier *Flow Informations* genannt) vorgestellt, die für jede WCET-Anwendung nutzbar sein soll. Diese definiert *Scopes* auf Ebene des Kontrollflussgraphen. Scopes fassen Basisblöcke oder wiederum Scopes zusammen, die über den *Header Node* der Scope höchstens einmal im Kontrollfluss erreicht werden können. Flow Informations haben die Form:

  Scope, ContextSpec, Constraint.

  Scope spezifiziert dabei die betroffene Scope, ContextSpec die Iterationen der Scope, in denen die Bedingung gelten muss und Constraint die Bedingung als Ungleichung über Ausführungshäufigkeiten von Basisblöcken, Folgen von Basisblöcken oder Kanten zwischen Basisblöcken, die alle innerhalb der Scope liegen müssen.

Den vorgestellten Ansätzen ist gemein, dass sich diese bei der Formulierung von Flow Facts im Wesentlichen auf einzelne Anweisungen beziehen, während ihre interne Darstellung – so weit bekannt – die Kanten des Kontrollflussgraphen zur Formulierung von Flow Facts nutzt.

Allerdings unterstützen alle Ansätze nur eingeschränkt Optimierungstechniken von Compilern. Insbesondere für die flexiblen und ausdrucksstarken Flow Informations erweisen sich die Datenaktualisierungen, die durch Optimierungstechniken notwendig sind, als komplex.

4.4 Designanforderungen

Die Anforderungen an das Design von Flow Facts im WCC werden im Folgenden kurz dargestellt.

- Annotationen auf Basis des Quellcodes

  Da ein Benutzer des WCCs Flow Facts intuitiv aus dem Quellcode heraus bestimmen möchte, sollten diese auch mit Bezug zum Quellcode annotierbar sein, statt vom Benutzer transformiert und mit Bezug auf dessen Assemblercode annotiert werden zu müssen.

- Annotationen im Quellcode

  Um für den Benutzer eine sichtbare und intuitiv handhabbare Bindung von Flow Facts an den Quellcode zu erreichen, ist deren Annotation im Quellcode statt in einer eigenen Spezifikationsdatei wünschenswert. Dadurch darf der Quellcode aber nicht für dritte Compiler unbrauchbar werden.

- Ausdrucksstarke Annotationen

  Nicht nur grundlegende Begrenzungsinformationen sollen annotierbar sein, sondern auch Informationen, die eine qualitativ hochwertige – d. h. eine möglichst präzise – WCET-Abschätzung unterstützen. Zudem dürfen keine C Sprachkonstrukte ausgeschlossen werden. Es müssen z. B. direkte und indirekte Rekursionen und Schleifen per *goto*-Anweisung annotierbar sein.

- Einheitliche Annotationen

  Die Annotationen sollen durch einen möglichst einheitlichen Mechanismus erfolgen.

- Maschinenkompatibel

  Da in einer späteren Ausbaustufe des WCCs implizite Flow Facts automatisch annotiert werden sollen (insbesondere jene impliziten Flow Facts, die aiT nicht eigenständig bestimmen kann), müssen Flow Facts durch entsprechende Analysetechniken jederzeit generierbar sein.

- Updatemechanismen unterstützen

  Neben der Spezifikation der Flow Facts ist es wichtig, Mechanismen zum effektiven Zugriff und zur Manipulation zu entwickeln, da diese durch die Optimierungen des Quellprogramms notwendig werden.

- Integration in ICD-C und ICD-LLIR

  Da Flow Facts eine Erweiterung der Intermediate Representations des WCCs sein werden, ist eine homogene Integration in die vorhandene ICD-C bzw. ICD-LLIR anzustreben.

- Kompatibel zu aiT

  Flow Facts müssen so gestaltet sein, dass sie letztendlich in eine Darstellung für aiT übersetzt werden können.

Da das Interface zu aiT fest vorgegeben ist, ist dies näher zu untersuchen.

Interface aiT

Für aiT sind folgende für Flow Facts wesentliche Annotationen möglich:

- Unausführbarer Code

  Eine Anweisung wird als nicht ausführbar gekennzeichnet.

- Wert einer Bedingung

  Eine Bedingung, z. B. die einer *if*-Anweisung, wird als stets wahr oder stets falsch spezifiziert.

- Rekursionsbeschränkung

  Für die WCET-Analyse wird eine Rekursion begrenzt, indem die maximale Anzahl von Aufrufen der rekursiven Funktion begrenzt wird.

- Schleifeniterationsgrenzen (*Loop Bounds*)

  Für eine Schleife, für alle Schleifen einer Routine oder global für alle Schleifen eines Programms können Minimum und Maximum der Iterationshäufigkeit annotiert werden.

- Kontrollflussbeschränkungen (*Flow Constraints*)

  Eine Einschränkung des Kontrollflusses ist durch lineare Ungleichungen möglich, die die Ausführungshäufigkeiten von Anweisungen (mit Gewichtungsfaktoren) zueinander in Relation setzen.

Alle Annotationen für aiT spezifizieren die Ausführungshäufigkeiten von Anweisungen näher, Ausführungshäufigkeiten von Übergängen zwischen Anweisungen (genauer: Kanten im Kontrollflussgraph) sind nicht direkt spezifizierbar.

4.5 Design der Flow Facts

Bei den Voraussetzungen und Anforderungen sind drei wesentliche Unterschiede zu den Ansätzen in der Literatur festzuhalten:

- Optimierungen eines Compilers sind zu unterstützen.
- Flow Facts müssen abschließend für ein externes Tool formuliert werden.
- Flow Facts sollen in vorhandene Datenstrukturen integriert werden (in Literatur oft parallele Haltung).

Basis von Flow Facts

Wie in vorangegangenen Arbeiten üblich, ist es sinnvoll, Flow Facts in Abhängigkeit von Programmpunkten durch den Benutzer formulieren zu lassen, da diese im Quellcode verfügbar sind (jede Anweisung dort stellt zunächst einmal einen Programmpunkt dar), während Übergänge zwischen den Programmpunkten nur implizit durch die Semantik der Anweisungen definiert und so schwer für einen Programmierer und Nutzer des Compilers zu fassen sind.

Durch Nutzung des externen Tools aiT zur Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ ist es bei der Übergabe der Flow Facts wiederum nötig, diese in Abhängigkeit von Programmpunkten – dort in

Form von Basisblöcken – zu formulieren. Da zudem keine Kontrollflussdarstellung dauerhafter Bestandteil von ICD-C ist, liegt es nahe, Flow Facts im gesamten Compiler auf Basis von Programmpunkten zu modellieren.

Dies entspricht nicht – soweit bekannt – dem Ansatz vorheriger Arbeiten. Das liegt daran, dass der Kontrollfluss über seine Kanten mehr Möglichkeiten zur Beschreibung von Flow Facts bietet und insgesamt eine genauere Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ ermöglicht. Dennoch ist die Modellierung der Flow Facts auf Basis der Programmpunkte im WCC sinnvoll und nicht einschränkend.

Der höchste Grad an Genauigkeit zur Beschreibung von Flow Facts wird durch die Schnittstelle des externen Analysetools vorgegeben, die – wie beschrieben – bei aiT auf Programmpunkten basiert. Die compilerinterne Nutzung eines voll kompatiblen Designs verhindert, dass bei der Konvertierung eines alternativen internen Modells für die Schnittstelle wichtige Informationen verloren gehen.

Bei einem gemeinsamen Modell liegt es z. B. bei den Analysemechanismen, ihre Ergebnisse kompatibel zu annotieren. Da Analysen i. d. R. über zusätzliches Wissen zum zu annotierenden Flow Fact verfügen, sind sie optimal in der Lage, eine möglichst verlustfreie Konvertierung durchzuführen. Viele Analysen werden gar auf dem Flow Fact Modell des WCCs arbeiten, so dass keine Konvertierungen (mit Genauigkeitsverlust) notwendig werden.

Vor allem aber müssen die Annotationen des Benutzers nicht dreimal (verlustbehaftet) konvertiert werden, nämlich bei der Extraktion aus dem Quellcode, zur Darstellung für die Schnittstelle zu aiT sowie in aiT. Es verbleibt bei einem zu aiTs Schnittstelle kompatiblen Modell nur eine Konvertierung in aiT.

Deshalb sind Flow Facts im WCC auf Basis von Programmpunkten zu entwerfen. Dies sind im Quellcode Anweisungen, in ICD-C Statements und in der ICD-LLIR Basisblöcke. Auf dieser Basis werden zwei Arten von Flow Facts entwickelt: *Flowrestrictions* und *Loopbounds*.

Flowrestrictions und Loopbounds

Flowrestrictions ermöglichen es, die Ausführungshäufigkeiten beliebiger Anweisungen gewichtet zueinander ins Verhältnis zu setzen. Diese Möglichkeit allein ist ausdrucksstark und erlaubt, alle wesentlichen Kontrollflussinformationen, die auch anschließend durch aiT nutzbar sind, zu spezifizieren. Dabei entsprechen Flowrestrictions im Wesentlichen aiTs linearen Flow Constraints.

Die Flow Constraints weisen gegenüber den Flowrestrictions jedoch eine Besonderheit auf. Sie können mit einem *Qualifier* versehen werden. Der Qualifier EACH drückt dabei aus, dass bei jedem möglichen Programmablauf das entsprechende Flow Constraint zu erfüllen ist, der Qualifier SUM dagegen, dass nur für die Summe über alle möglichen Programmabläufe das Flow Constraint erfüllt sein muss. Die Unterscheidung in verschiedene Programmabläufe wird dabei durch aiTs Konzept der Kontexte für eine genauere WCET-Analyse erreicht. Den Unterschied zwischen den Qualifiern verdeutliche folgendes Beispiel.

Beispiel: *Gibt es für ein Flow Constraint* flow each A <= B; *zwei verschiedene Kontexte, so gilt in beiden Kontexten, dass die Ausführungshäufigkeit der Anweisung* A *kleiner gleich*

der Ausführungshäufigkeit der Anweisung B *sein muss. Dies entspricht i. d. R. der intuitiv beabsichtigten Aussage.*

Wiederum für zwei Kontexte erlaubt das Flow Constraint flow sum A <= B; *hingegen in einem Kontext durchaus eine größere Ausführungshäufigkeit für* A *als für* B*, so lange die Summe der Ausführungshäufigkeiten von* A *über beide Kontexte kleiner gleich der Summe der Ausführungshäufigkeiten von* B *ist. Dies führt zu einer schwächeren Aussage als die Nutzung des Qualifiers* EACH *und damit u. U. zu einer höheren* $WCET_{est}$*.*

Der WCC arbeitet unabhängig von Kontexten, und dementsprechend sind auch Annotationen unabhängig von Kontexten zu gestalten. Vom Benutzer ist eine Annotation im Sinne des intuitiven EACH Qualifiers zu erwarten. Da aiT den EACH Qualifier allerdings nur für Flow Constraints erlaubt, deren Elemente alle in einer Routine liegen, ist im Zweifelsfall der Qualifier SUM bei der Übersetzung der Flowrestrictions in Flow Constraints zu nutzen. Dies kann zu einer zusätzlichen Überabschätzung der $WCET_{est}$ führen. Diese Abschätzung ist aber zumindest sicher.

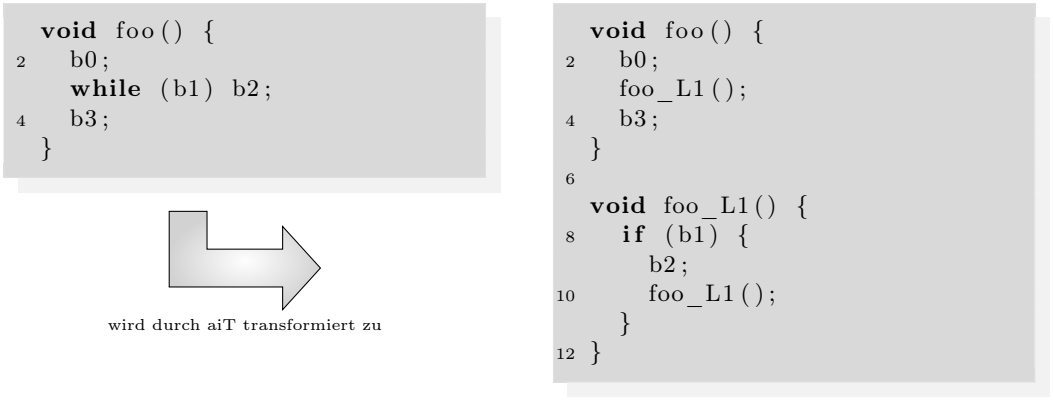


Abbildung 4.1: Bsp. einer Schleifentransformation durch aiT nach [AbsInt 2006]

Insbesondere bei der Begrenzung von Schleifeniterationen kann sich dies jedoch negativ auswirken, vor allem, da durch aiTs Schleifentransformation – wie ein Beispiel in Abb. 4.1 veranschaulicht – Schleifen stets in eine eigene Routine gekapselt werden. Somit ist der Qualifier SUM zwingend. Da aber gerade eine präzise Einschränkung der Iterationshäufigkeit von Schleifen bedeutend für die Qualität der $WCET_{est}$-Berechnung ist – Programme verbringen einen Großteil ihrer Ausführungszeit in Schleifen –, werden die Flowrestrictions um Loopbounds ergänzt. Die Loopbounds des WCCs können für aiT als dessen Loop Bounds spezifiziert werden, die eine präzisere Analyse als dessen Flow Constraints ermöglichen.

Loopbounds erlauben für Schleifen die Spezifikation einer oberen und einer unteren Grenze für deren Iterationshäufigkeit.

Spezifikation von Flow Facts im Quellcode

Flow Facts werden gemäß der Designanforderung im Quellcode annotiert. Um aber weder die Syntax der Sprache zu ändern, noch den Code auf andere Weise für alternative Compiler

unbrauchbar zu machen (wie z. B. wcetC), werden Pragmas zur Annotierung genutzt (vgl. Abschnitt 3.1.1).

Pragmas erlauben, Anweisungen an einen Compiler für seine Übersetzungsarbeit zu geben. So kann bspw. einem Compiler mitgeteilt werden, dass eine Bedingung einer Schleife i. d. R. wahr ist. Dieser kann entsprechend den Assemblercode für eine wahre Aussage optimieren.

Es gibt kein einheitliches Muster für Pragmas. Da aber ein Compiler, der ein Pragma nicht kennt, dieses ignorieren muss [ISO 9899 1999], kann zur Annotierung der Flow Facts die Menge der vom WCC verstandenen Pragmas erweitert werden, ohne die Übersetzung durch andere Compiler zu behindern. Zufällige Überschneidungen der Pragmamengen können natürlich nicht ausgeschlossen werden.

Als Pragma wird eine Annotation per

#pragma ANNOTATION LINEBREAK

oder per

_Pragma("ANNOTATION")

im Quellcode eines Programms spezifiziert.

ANNOTATION	⊨ MARKER | FLOWRESTRICTION | LOOPBOUND

Tabelle 4.1: EBNF Annotation

Es werden Pragmas zur Kennzeichnung von Anweisungen in Abschnitt 4.5.1, für Flowrestrictions in Abschnitt 4.5.2 und für Loopbounds in Abschnitt 4.5.3 eingeführt. Ihre Grammatik wird als EBNF definiert, beginnend in Tabelle 4.1 mit dem Startnichtterminal *ANNOTATION*, und fortgesetzt in den Tabellen 4.2 - 4.4 für Marker, Flowrestrictions und Loopbounds.

4.5.1 Hilfsannotationen

Anweisungen im Quellprogramm werden durch *Marker* mit symbolischen Namen (*Identifier*, siehe Tabelle 4.2) versehen, so dass sie über diese für den Programmierer zur Beschreibung (un)ausführbarer Pfade referenzierbar sind. Marker gehören nach Abschnitt 4.1 zur Klasse der Hilfsannotationen.

MARKER	⊨ marker NAME
NAME	⊨ *Identifier*

Tabelle 4.2: EBNF Marker

Die Abbildung der als Pragma spezifizierten Marker auf Anweisungen erfolgt durch die Anbindung von Pragmas an Statements und Expressions durch den ICD-C Parser. Die Marker einer Expression werden dabei deren Eltern Statement zugeordnet. Einige Besonderheiten sind zu beachten:

- Anweisungsblöcke (Compound Statements in der IR) können nicht markiert werden. Dies ist keine Einschränkung der Aussagekraft von Flow Facts, weil Anweisungsblöcke keinen Einfluss auf den Kontrollfluss der Programmausführung haben.

  Da jedes Flow Fact für aiT mit Bezug auf Basisblöcke formuliert werden muss, ein Answeisungsblock aber durchaus mehrere Basisblöcke umfassen kann, verhindert ein Verzicht auf Markierungen von Anweisungsblöcken letztendlich Mehrdeutigkeiten bei der notwendigen Zuordnung dieser Markierungen zu einem Basisblock.

- Für jede Funktion (genauer dessen Top Compound Statement, das somit eine Ausnahme bzgl. der Annotierbarkeit von Anweisungsblöcken darstellt) wird automatisch ein Marker mit dessen Funktionsnamen bereit gestellt. Dieser wird später dem Basisblock des Funktionseinstiegs zugeordnet.

- Markierungen von Schleifen beziehen sich immer auf deren Abbruchbedingung. Entsprechend können *for*-Anweisungen ohne Abbruchbedingung nicht markiert werden. Bei diesen *for*-Anweisungen verbleiben noch die Anweisungen im Schleifenkörper sowie dessen Folgeausdruck (der nach jeder Iteration des Schleifenkörpers ausgeführt wird) zur Markierung und zur Formulierung von Flow Facts.

  Zu beachten ist, dass die Abbruchbedingung einer Schleife mit n Iterationen für eine *for*- und *while*-Anweisung $n + 1$ mal, für eine *do-while*-Anweisung n mal ausgeführt wird.

Im WCC werden die Identifier statt durch alphanumerische Zeichenfolgen durch Zeiger auf das entsprechende Element der Datenstruktur dargestellt. Dies erlaubt nicht nur einen direkten Zugriff auf das betroffene Element, sondern bietet vor allem eine eineindeutige Abbildung von Identifiern auf Elemente und zudem einen effizienten Vergleich von Identifiern.

Semantisch wird durch die Marker bzw. Zeiger kein Bezug auf eine Anweisung, ein Statement bzw. einen Basisblock genommen, sondern auf eine Entscheidungsvariable, die die Ausführungshäufigkeit des entsprechenden Elementes auf einem Kontrollflusspfad beschreibt. Der Wertebereich der Entscheidungsvariablen wird u. a. durch Flowrestrictions und Loopbounds eingeschränkt.

4.5.2 Flowrestrictions

Flowrestrictions erlauben, die Ausführungshäufigkeit verschiedener Elemente einer Compilation Unit gewichtet zueinander ins Verhältnis zu setzten. Die Einschränkung auf Elemente einer Compilation Unit entsteht durch die separate WCET-Analyse für jede Compilation Unit bzw. LLIR im WCC. Bei einer (zukünftig geplanten) gemeinsamen Analyse wird diese Einschränkung entfallen. Der Aufbau von Flowrestrictions wird in Tabelle 4.3 definiert.

FLOWRESTRICTION	⊨ flowrestriction SIDE COMPARATOR SIDE
COMPARATOR	⊨ >= | <= | =
SIDE	⊨ SIDE + SIDE | NUM * REFERENCE
NUM	⊨ *Non negative Integer*
REFERENCE	⊨ *Identifier* | *Functionname*

Tabelle 4.3: EBNF Flowrestriction

Die Referenzen auf Anweisungen (*Identifier*) müssen durch Marker erzeugt werden. Zusätzlich existiert – wie oben beschrieben – stets ein Marker für jede Funktion mit deren Namen (*Functionname*). Als Gewichtungsfaktoren sind nicht negative ganze Zahlen ($\mathbb{N}_0$) erlaubt (*Non negative Integer*).

Aussagekraft

Mittels Flowrestrictions lassen sich alle Arten von Schleifen begrenzen, insbesondere auch Multi-Entry Loops – also Schleifen mit mehr als einer Möglichkeit betreten zu werden – und darüber hinaus Schleifen, die nicht direkt als solche modelliert sind – z. B. per *goto*-Anweisung erzeugte Schleifen. Wird eine Schleife maximal *max* mal iteriert, so wird dazu ein Marker *markeraußen* auf eine Anweisung außerhalb der Schleife und ein Marker *markerinnen* auf eine Anweisung innerhalb der Schleife benötigt. Idealerweise sind dies Anweisungen, die genau einmal ausgeführt werden (z. B. *if*-Anweisungen oder arithmetische Ausdrücke und Zuweisungen), da ansonsten der Wert von *max* zu modifizieren ist. Zudem darf sich zwischen den Markern und der Schleifenabbruchbedingung keine Sprunganweisung (*goto*-Anweisung, *break*-Anweisung, ...) und kein Sprungziel (Label, *case*-Anweisung, ...) befinden, da ansonsten der Kontrollfluss wesentlich beeinflusst wird. Die Schleifeniterationshäufigkeit lässt sich dann durch

```
flowrestriction 1*markerinnen <= max*markeraußen
```

beschränken.

```
_Pragma("marker markeraußen")
Anweisung A;

for( i = 0; i < 10; i++ )
  for( j = i; j < 10; j++ )
    _Pragma("marker markerinnen")
    Anweisungsblock B

_Pragma("flowrestriction 1*markerinnen <= 55*markeraußen")
```

Der Anweisungsblock B verändert die Variablen i und j nicht.
Beispiel nach [Ermedahl u. a. 2002].

Abbildung 4.2: Triangulierte Schleife

Besonders geeignet ist dieses Verfahren für *triangulierte Schleifen* wie in Abb. 4.2. Mit einem Marker *markeraußen* auf eine Anweisung A außerhalb beider Schleifen und einem Marker *markerinnen* auf eine Anweisung im Anweisungsblock B (sonst analoge Bedingungen zu oben) lässt sich die Iterationshäufigkeit per

```
flowrestriction 1*markerinnen <= 55*markeraußen
```

präzise beschränken, während Loopbounds nur für beide Schleifen eine Beschränkung auf je 10 Iterationen – und damit 100 Ausführungen für den Anweisungsblock B – ermöglichen würden.

Rekursionen lassen sich analog zu Schleifen mit einem Marker *markercall* auf die die Rekursion aufrufende Anweisung (entspricht in ICD-C einem Expression Statement) und dem

Funktionsnamen *recfunc* der rekursiven Funktion begrenzen. Für eine rekursive Funktion mit maximal *max* Ausführungen pro externen Aufruf beschränkt dann

```
flowrestriction 1*recfunc <= max*markercall
```

die Rekursion.

Weitere nahezu beliebige Relationen, wie z. B. die Exklusivität von Basisblöcken, lassen sich durch Flowrestrictions beschreiben.

```
  if( cond )
2   x = true; //Anweisung A

4 for( ... )
    if( x ) Anweisungsblock B;
```

Die Ausführung von Anweisung A impliziert die Ausführung des Anweisungsblocks B. Beispiel nach [Ermedahl u. a. 2002].

Abbildung 4.3: Tief verschachtelte Abhängigkeit

Nicht formulieren lassen sich hingegen *tief verschachtelte Abhängigkeiten* wie in Abb. 4.3. Allerdings sind diese auch nicht für aiT spezifizierbar.

Mathematisches Modell

Die vorgestellten Flowrestrictions lassen sich in folgendes Modell überführen:

Sei X die Menge der Entscheidungsvariablen, die den Anweisungen ihre Ausführungshäufigkeit im Programmablauf zuordnen, mit $|X| = n$ Elementen und der Indexmenge $J = \{1, \ldots, n\}$. A und B seien Matrizen der Form

$$A = \begin{pmatrix} a_{1\,1} & \cdots & a_{1\,n} \\ \vdots & \ddots & \vdots \\ a_{m\,1} & \cdots & a_{m\,n} \end{pmatrix},\ B = \begin{pmatrix} b_{1\,1} & \cdots & b_{1\,n} \\ \vdots & \ddots & \vdots \\ b_{m\,1} & \cdots & b_{m\,n} \end{pmatrix}. \tag{4.1}$$

Dann lassen sich m Flowrestrictions darstellen als:

$$A \cdot x \leq B \cdot x \tag{4.2a}$$

mit

$$\forall i, j \text{ mit } i \in \{1, \ldots, m\}, j \in J \text{ gilt:}$$
$$a_{i\,j}, b_{i\,j} \in \mathbb{N}_0 \tag{4.2b}$$
$$a_{i\,j} > 0 \Rightarrow b_{i\,j} = 0 \tag{4.2c}$$
$$b_{i\,j} > 0 \Rightarrow a_{i\,j} = 0 \tag{4.2d}$$
$$x_j \in \mathbb{N}_0 \tag{4.2e}$$

Diese Form wird durch folgende Schritte für jede der m Flowrestrictions erreicht:

1. Bringe alle Summanden auf eine Seite der (Un-)Gleichung.

2. Fasse zusammen.

   Dies garantiert, dass jedes x_j höchstens einmal pro Flowrestriction vorkommt, und erfüllt somit (4.2c) und (4.2d).

3. Bringe Summanden mit negativem Faktor auf die andere Seite der (Un-)Gleichung.

   Dies wiederum garantiert, dass alle $a_{i\,j}$ und $b_{i\,j}$ nicht negativ sind, und erfüllt so (4.2b).

4. Bringe (Un-)Gleichung in die „kleiner gleich Form“ unter Nutzung folgender Identitäten:

$$A \cdot x \leq B \cdot x \quad \Leftrightarrow \quad A \cdot x \leq B \cdot x \tag{4.3}$$
$$A \cdot x \geq B \cdot x \quad \Leftrightarrow \quad B \cdot x \leq A \cdot x \tag{4.4}$$
$$A \cdot x = B \cdot x \quad \Leftrightarrow \quad A \cdot x \leq B \cdot x \ \wedge \ B \cdot x \leq A \cdot x \tag{4.5}$$

   Durch diese Umformung wird die Form in (4.2a) erfüllt.

Entsprechend diesem Modell wird die Datenstruktur der Flowrestrictions in Abschnitt 4.6 modelliert werden.

4.5.3 Loopbounds

Loopbounds erlauben, für jede Ausführung einer Schleife deren minimale und maximale Iterationshäufigkeit zu spezifizieren. Der Aufbau von Loopbounds wird in Tabelle 4.4 definiert.

LOOPBOUND	⊨ loopbound min NUM max NUM
NUM	⊨ *Non Negative Integer*

Tabelle 4.4: EBNF Loopbound

Zur Zuordnung einer Loopbound zu einer Schleife wird kein Marker benötigt, da sich diese auf genau eine Schleifenanweisung bezieht. Die Zuordnung erfolgt durch die Zuordnung des spezifizierenden Pragmas zu einer Anweisung. Diese Anweisung muss eine *for*-, *while*- oder *do-while*-Anweisung sein. Dementsprechend lassen sich durch Loopbounds keine impliziten Schleifen – z. B. durch die *goto*-Anweisung erzeugte Schleifen – in ihrer Iterationshäufigkeit beschränken. Darüber hinaus können keine *for*-Anweisungen ohne Abbruchbedingung (unendliche Schleife) per Loopbound annotiert werden. Implizite Schleifen und *for*-Anweisungen ohne Abbruchbedingung sind durch Flowrestrictions zu annotieren.

Aussagekraft

Da die Loopbounds des WCCs für aiT als dessen Loop Bounds annotiert werden, unterliegen sie dessen Einschränkungen. So dürfen für aiT nur solche Schleifen per Loop Bound annotiert werden, die bei dessen Schleifentransformation erkannt werden. Dies sollte i. d. R. für *for*- (mit Abbruchbedingung), *while*- und *do-while*-Schleifen ohne mehrere Einstiegspunkte gelingen.

Vor allem Multi-Entry Loops, also Schleifen mit mehreren Einstiegspunkten, werden dagegen von aiT nicht erkannt und dürfen dementsprechend auch nicht per Loop Bound annotiert werden. Wird dennoch eine nicht erkannte Schleife per Loop Bound annotiert, bricht aiT im

besten Fall mit einer Fehlermeldung ab, im schlimmsten Fall bezieht es diese Information auf eine andere Schleife und berechnet dadurch unter Umständen eine nicht sichere $\mathrm{WCET}_{\mathrm{est}}$.

Daher ist die Nutzung von Loopbounds im WCC einzuschränken, um möglichst sicherzustellen, dass die annotierten Schleifen auch stets von aiT erkannt werden. Und genau deswegen dürfen nur *for*- (mit Abbruchbedingung), *while*- und *do-while*-Schleifen per Loopbound annotiert werden. Es wird zudem empfohlen, bei Schleifen mit Label Statements im Schleifenkörper oder *case*- bzw. *default*-Statements ohne ebenfalls umgebendes *switch*-Statement im Schleifenkörper auf die Annotierung durch Loopbounds zu verzichten, da diese Schleifen möglicherweise nicht erkannt werden. Die Schleifen können alternativ per Flowrestriction annotiert werden.

Mathematisches Modell

Loopbounds lassen sich teilweise in die Form des Modells in (4.2) überführen:

Sei $x_j \in X$ eine Entscheidungsvariable für eine Schleifenanweisung (in ICD-C – da einem Loop Statements selbst keine Entscheidungsvariable zugeordnet werden kann – die der Abbruchbedingung des Loop Statements, in ICD-LLIR die des Basisblocks, der den Sprungbefehl der Schleife beinhaltet) mit einer Loopbound, die für diese Schleife mindestens *min* und höchstens *max* Iterationen spezifiziert. Sei $\hat{X} \subset X$ die Menge der Entscheidungsvariablen der unmittelbaren Vorgänger der Schleife, d. h. die Menge der Entscheidungsvariablen aller Anweisungen, Statements bzw. Basisblöcke, von denen mindestens ein Nachfolger ein Element der Schleife ist, und die selbst nicht zur Schleife gehören. Dann gilt:

$$1 \cdot x_j \leq \sum_k max \cdot x_k \qquad \forall k : x_k \in \hat{X} \tag{4.6}$$

für eine Schleife mit Prüfung der Abbruchbedingung zum Ende jeder Iteration, bzw.

$$1 \cdot x_j \leq \sum_k (max + 1) \cdot x_k \qquad \forall k : x_k \in \hat{X} \tag{4.7}$$

für eine Schleife mit Prüfung ihrer Abbruchbedingung zu Beginn jeder Iteration.

Somit wird die für die WCET-Analyse wichtige obere Schranke der Iterationshäufigkeit in eine Flowrestriction transformiert. Eine entsprechende Annotierung durch den Benutzer ist jedoch potentiell genauer. So können durch Flowrestrictions u. U. bei Multi-Entry Loops für jeden Einstiegspunkt eigene Iterationshäufigkeiten spezifiziert werden.

Die untere Schranke lässt sich insbesondere in der ICD-LLIR nicht analog zu (4.6) bzw. (4.7) in

$$\sum_k min \cdot x_k \leq 1 \cdot x_j \qquad \forall k : x_k \in \hat{X} \tag{4.8}$$

transformieren, da die Vorgänger unter Umständen mehrere Nachfolger haben, also nicht zwangsläufig deren Ausführung zu einer Ausführung der Schleife führt.

```
;Assembler Code
2
...
4
.pred:
6   jge(D11, D8, .exit);

8 .loop:
    ...
10  jge(D11, D8, .loop);
```

```
;Assembler Code Fortsetzung
12
...
14
.addEntry:
16  jge(D9, D8, .loop);

18 .exit:
    ret();
20
```

Abbildung 4.4: Bsp. für die Transformation einer Loopbound in eine Flowrestriction

Beispiel: *Die Assemblercodesequenz in Abb. 4.4 enthält eine Schleife* .loop *mit den zwei Vorgängern* .pred *und* .addEntry *und Prüfung ihrer Abbruchbedingung zum Ende einer Iteration (Abb. 4.4, Zeile 10). Die Schleife sei durch eine Loopbound annotiert, die für diese mindestens drei, höchstens jedoch fünf Iterationen pro Ausführung ausweist (*loopbound min 3 max 5*).*

Nach (4.6) *lässt sich das Maximum* 5 *zur grundlegenden Begrenzung der Schleife per Flowrestriction wie folgt nutzen:* 1*.*loop* <= 5*.*pred* + 5*.*addEntry*. *Dies ist sicher, jedoch nicht präzise, da die Ausführungen von* .pred *und von* .addEntry *nicht zwangsläufig in einer Ausführung der Schleife münden müssen.*

Aus demselben Grund ist jedoch das Nutzen des Minimums nicht möglich, denn eine Flowrestriction nach (4.8)*:* 3*.*pred* + 3*.*addEntry* <= 1*.*loop* *besagt, dass (zumindest die ersten beiden) Ausführungen von* .pred *und* .addEntry *in Ausführungen der Schleife münden. Der mögliche Kontrollfluss einer einmaligen Ausführung von* .pred *mit Sprung nach* .exit *ohne Ausführung von* .loop *und* .addEntry *führt zu folgender, falscher Aussage:* $3 * 1 + 3 * 0 \leq 1 * 0$ *und würde somit durch die Flowrestriction verboten werden, ohne dass er vorher durch die Loopbound verboten gewesen wäre. Daher ist diese Transformation nicht sicher.*

Um Loopbounds präzise zu fassen, ist folglich die Entwicklung eines eigenen Modells notwendig:

Sei $\tilde{X}$ die Menge der Entscheidungsvariablen, die den – in aiTs Schleifentransformation erkannten – Schleifen ihre Iterationshäufigkeit bei einmaliger Ausführung zuordnen. Dann lässt sich eine Loopbound mit den Werten $min_{annotiert}$ und $max_{annotiert}$ für eine Schleife l mit der Entscheidungsvariablen $\tilde{x}_l \in \tilde{X}$ darstellen als:

$$min_{annotiert} \leq \tilde{x}_l \leq max_{annotiert} \tag{4.9a}$$

mit

$$min_{annotiert}, max_{annotiert} \in \mathbb{N}_0 \tag{4.9b}$$
$$\tilde{x}_l \in [min_l, max_l] \tag{4.9c}$$
$$min_l, max_l \in \mathbb{N}_0 \tag{4.9d}$$

Der Wert einer Entscheidungsvariablen $\tilde{x}_l \in \tilde{X}$ lässt sich nur durch ein Intervall bescheiben, da aiT für die verschiedenen Ausführungen einer Schleife durch sein Konzept der Kontexte durchaus verschiedene Iterationshäufigkeiten ermitteln kann (während die Ausführungshäufigkeit für jedes $x \in X$ auf dem $\mathrm{WCET}_{\mathrm{est}}$-Pfad eindeutig ist und daher genau einer nicht negativen Ganzzahl entspricht). So werden der Entscheidungsvariablen $\tilde{x}_l$ u. U. mehrere Ergebnisse zugeordnet, die sich nur durch ein Intervall gemeinsam beschreiben lassen.

Dass aiT die Entscheidungsvariablen aus $\tilde{X}$ nicht als Iterationshäufigkeit einer Schleife, sondern als Anzahl der Tests der Abbruchbedingung der Schleife bei einer Ausführung interpretiert, darf bei dieser Modellbildung vernachlässigt werden, da die Loopbounds vor der Spezifikation für aiT automatisch angepasst werden.

Für Loopbounds ist folglich nicht nur eine eigene Datenstruktur zu entwickeln, um sie später für aiT als dessen Loop Bounds spezifizieren zu können, sondern in Kapitel 5 sind für Flowrestrictions und Loopbounds z. T. separate Transformationen zu entwickeln, um den jeweiligen Eigenheiten der Modelle gerecht zu werden.

4.6 Datenstruktur

Zur Darstellung der Flow Facts in ICD-C und ICD-LLIR wurden neue Datenstrukturen entwickelt. Eine Übersicht dieser Neuentwicklung für ICD-C ist in Abb. 4.5 auf der nächsten Seite zu sehen. Die Klassen der ICD-LLIR wurden analog entwickelt, sind allerdings an LLIR_BB-Objekte statt IR_Stmt-Objekte und an ein LLIR-Objekt statt eines IR-Objekts angebunden.

Die Beschreibungen für ICD-C im Folgenden gelten auch für die ICD-LLIR, auf Unterschiede wird explizit hingewiesen. Die vorgestellten Strukturen werden dabei i. d. R. ohne das Präfix IR_ bzw. LLIR_ benutzt, sofern eine Unterscheidung zwischen IR und LLIR eindeutig oder nicht notwendig ist.

Datenkapselung

Allen Memberdaten der neuen Datenstrukturen ist gemein, dass kein direkter Zugriff von außen auf sie ermöglicht wird. Der Zugriff wird über entsprechende set()- und get()- sowie Manipulationsmethoden geregelt. Auch dabei gilt, dass eine get()-Methode keinen direkten Zugriff auf die internen Daten erlaubt, sondern eine Kopie dieser Daten zurückliefert. Dies erfordert zwar erhöhten Aufwand, i. d. R. werden diese Methoden aber vor allem Dank der Verwaltung von Referenzen durch IR_Flowfactref-Objekte (s. u.) selten benötigt. Andererseits kann so die Datenkonsistenz sichergestellt werden.

Flow Facts

Entsprechend der beiden Arten von Flow Facts gibt es eine Klasse IR_Flowrestriction, die die gewichteten Ausführungshäufigkeiten von Statements zueinander ins Verhältnis setzt, sowie eine Klasse IR_Loopbound, die für ein IR_LoopStmt seine minimale und maximale Iterationshäufigkeit verwaltet. Da es in der LLIR keine entsprechend spezialisierten Datenstrukturen für Schleifen wie in ICD-C gibt, werden die Schleifeninformationen dem Basisblock mit der Schleifenabbruchbedingung zugeordnet.

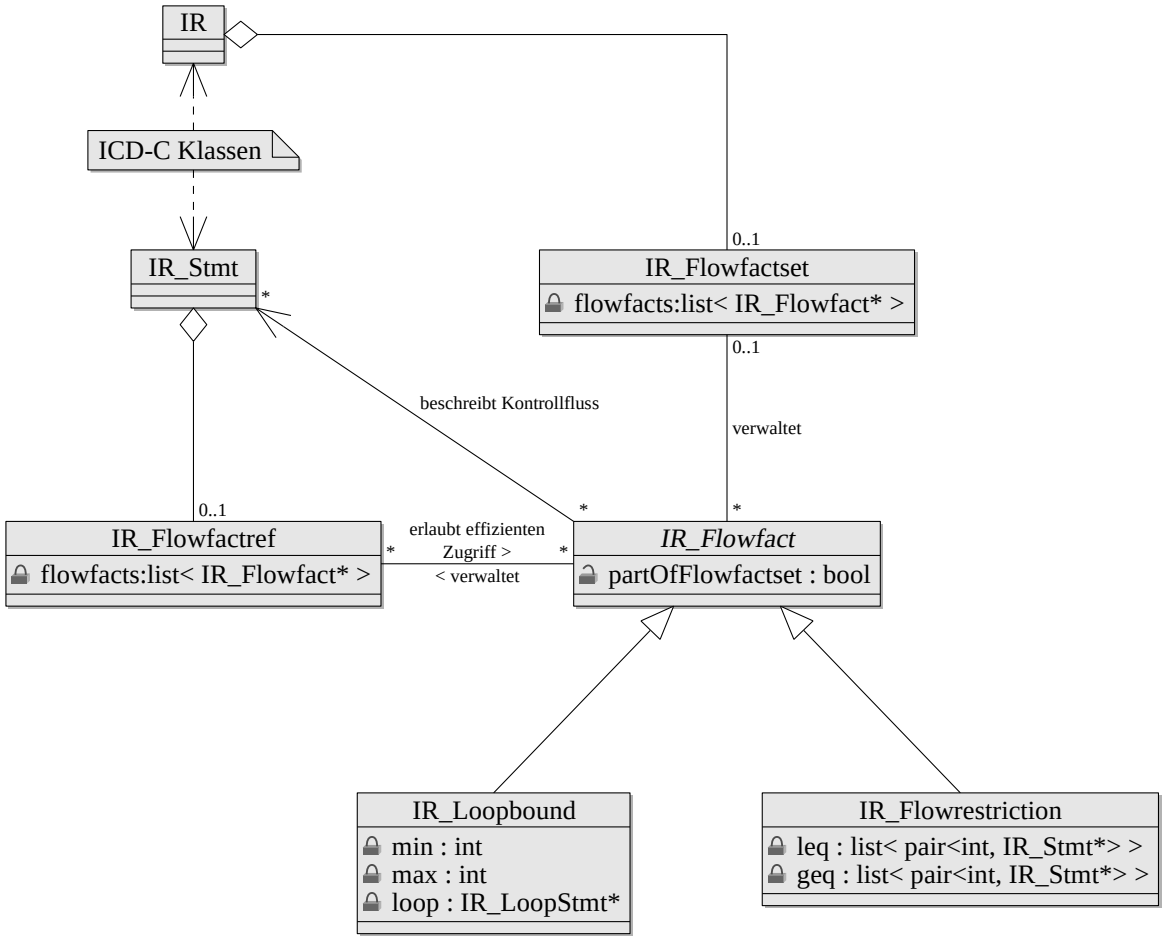


Abbildung 4.5: Übersicht Datenstrukturen

Die Flowrestrictions werden stets in der in Abschnitt 4.5.2 modellierten Form als Ungleichung gehalten und bieten daher zwei Datencontainer leq und geq für die Summanden der „kleiner gleich Seite“ und der „größer gleich Seite“. Ein Summand besteht aus einem ganzzahligen Faktor und einer Referenz (einem Zeiger auf ein IR_Stmt-Objekt). Dabei gilt, dass die Faktoren stets positiv sind (ist der Faktor eines Summanden Null, so wird der vollständige Summand zu Null vereinfacht) und jedes IR_Stmt-Objekt höchstens einmal pro Flowrestriction referenziert wird. Dies ist durch die Methoden zur Modifikation der Daten addToLeq(int factor, IR_Stmt *stmt) und addToGeq(int factor, IR_Stmt *stmt) sichergestellt, die dementsprechend die einzigen Methoden mit Schreibzugriffen auf diese Memberdaten sind.

Loopbounds verwalten neben ihrem IR_LoopStmt zwei Ganzzahlen für die minimale und die maximale Iterationshäufigkeit. Diese dürfen regulär nicht negativ sein. Beide Werte können jedoch gemeinsam auf -1 gesetzt werden, um eine Schleife als solche zu markieren ohne jedoch Iterationshäufigkeiten zu spezifizieren. Dies ist insbesondere für die LLIR interessant, da dort Schleifen nicht direkt durch ihr LLIR_BB-Objekt als solche erkannt werden können.

Bevor ein Element mit Flow Facts aus der IR oder LLIR wieder entfernt wird, muss sicher-

gestellt werden, dass es zuvor aus allen Flow Facts entfernt wurde. Da die Mechanismen zum Einfügen und zum Entfernen von Elementen der IR bzw. der LLIR nicht modifiziert werden sollten, kann keine automatische Überwachung erfolgen.

Abstraktion

Um zum Einen eine Menge von Mindestfunktionen zur Verwaltung von Flow Facts bereitzustellen und zum Anderen alle Flow Facts gemeinsam verwalten zu können, erben beide Flow Fact Arten von der abstrakten Klasse IR_Flowfact. Eine Ergänzung um weitere Flow Fact Arten – z. B. spezielle Daten zur Analyse des Kontrollflusses und so zur automatischen Ermittlung weiterer Flowrestrictions oder Loopbounds – ist durch Ableitung der Klasse IR_Flowfact möglich.

Verwaltung der Flow Facts

Um alle Flow Facts gemeinsam in die vorhandene Datenstruktur zu integrieren, wurde die Klasse IR_Flowfactset entwickelt. Diese ist über die IR erreichbar und verwaltet alle für diese IR relevanten Flow Facts.

Diese zentrale Verwaltung für Flow Facts durch ein Flowfactset ist sinnvoll, da zum Einen einige Mechanismen einen effektiven Zugriff auf die Menge aller Flow Facts – insbesondere zum Übersetzen von Flow Facts zwischen den Abstraktionsebenen im Compiler – benötigen, zum Anderen aber Flowrestrictions sich durch ihre Summanden auf eine Vielzahl von Statements beziehen können und hier keine eindeutige Zuordnung einer Flowrestriction zu einem Statement möglich wäre.

Nachteil dieser zentralen Flow Fact Verwaltung ist allerdings, dass, um für ein Statement alle zugehörigen Flow Facts zu finden, die komplette Flow Fact Menge durchsucht werden müsste. Dabei wird diese Aufgabe im Rahmen von Transformationen von Flow Facts regelmäßig notwendig sein, oft sogar mit dem Ergebnis, dass kein Flow Fact betroffen ist.

Effizienter Zugriff auf Flow Facts

Um trotz der zentralen Verwaltung von Flow Facts effizient von einem IR_Stmt-Objekt auf dessen Flow Facts zugreifen zu können – und so auch effizient zu bestimmen, ob überhaupt Flow Facts betroffen sind – wurde die Klasse IR_Flowfactref entwickelt. Für jedes Objekt, das durch Flow Facts des Flowfactsets betroffen ist, hält sie Referenzen auf diese Flow Facts.

Die Erzeugung und Verwaltung derer Daten erfolgt voll automatisch. Sobald ein Flow Fact zum Flowfactset hinzugefügt wurde, werden die Referenzen in den Flowfactrefs der betroffenen Statements auf dieses Flow Fact bereitgestellt und bei Modifikation des Flow Facts aktualisiert. Wird ein Flow Fact aus dem Flowfactset entfernt, so werden auch seine Referenzen aus den Flowfactrefs entfernt.

Die Verwaltung der Flowfactrefs erfolgt direkt durch die Flow Facts, die dank eines Flags partOfFlowfactset wissen, ob sie aktuell zum Flowfactset gehören und dementsprechend bei einer Änderung ihrer Memberdaten die Flowfactrefs aktualisieren müssen. Dieses Flag wird vom Flowfactset gesetzt, sobald ein Flow Fact diesem hinzugefügt wird, und bei dessen Entfernung gelöscht. Dabei löst das Setzen des Flags automatisch ein initiales Hinzufügen und

das Entfernen des Flags ein Löschen von Referenzen in den entsprechenden Flowfactrefs auf dieses Flow Fact aus.

Ausgabe von Daten

Um Informationen über Flow Facts ausgeben zu können, ist jede Klasse mit einer Methode write(ostream &str, int indent = 0) const versehen worden. So werden durch ein Flowfactset z. B. alle Flow Facts der Datenstruktur, durch ein Flowfactref nur jene des entsprechenden Statements ausgegeben.

Diese Methoden werden z. B. für eine ausführliche Protokolldatei genutzt, in der alle Flow Facts betreffende Informationen festgehalten werden. Die Protokolldatei wird dabei von den Klassen zur Transformation von Flow Facts verwaltet.

Implementierungsdetails

Konzeptionell sind Frontend- und Backend-Datenstruktur des Compilers unabhängig von Flow Facts. Um diese Unabhängigkeit auch weiterhin zu gewährleisten, wurden die Instanzen der Klassen IR_Flowfactset und IR_Flowfactref – über die die Flow Facts in die Datenstruktur eingebunden werden – in ICD-C als *User Data* (vgl. Abschnitt 3.1.1) eingebunden. Analog werden in der ICD-LLIR die Instanzen der Klassen LLIR_Flowfactset und LLIR_Flowfactref als *Objectives* (vgl. Abschnitt 3.1.2) eingebunden. Damit waren an ICD-C und der ICD-LLIR selbst keine Modifikationen notwendig.

5 Transformation von Flow Facts

Ein Quellprogramm wird in einem Compiler auf den verschiedenen Abstraktionsebenen durch verschiedene Datenstrukturen dargestellt, die sich durch ihr Design für spezielle Optimierungen und Analysen eignen. Im WCC sind dies – wie bereits in Kapitel 3 vorgestellt – ICD-C und ICD-LLIR sowie für das externe Tool aiT CRL2. Bei der Übersetzung eines Quellprogramms in diese und zwischen diesen Datenstrukturen, aber auch durch Optimierungen des Quellprogramms auf diesen Datenstrukturen wird dessen Aufbau und Kontrollfluss modifiziert.

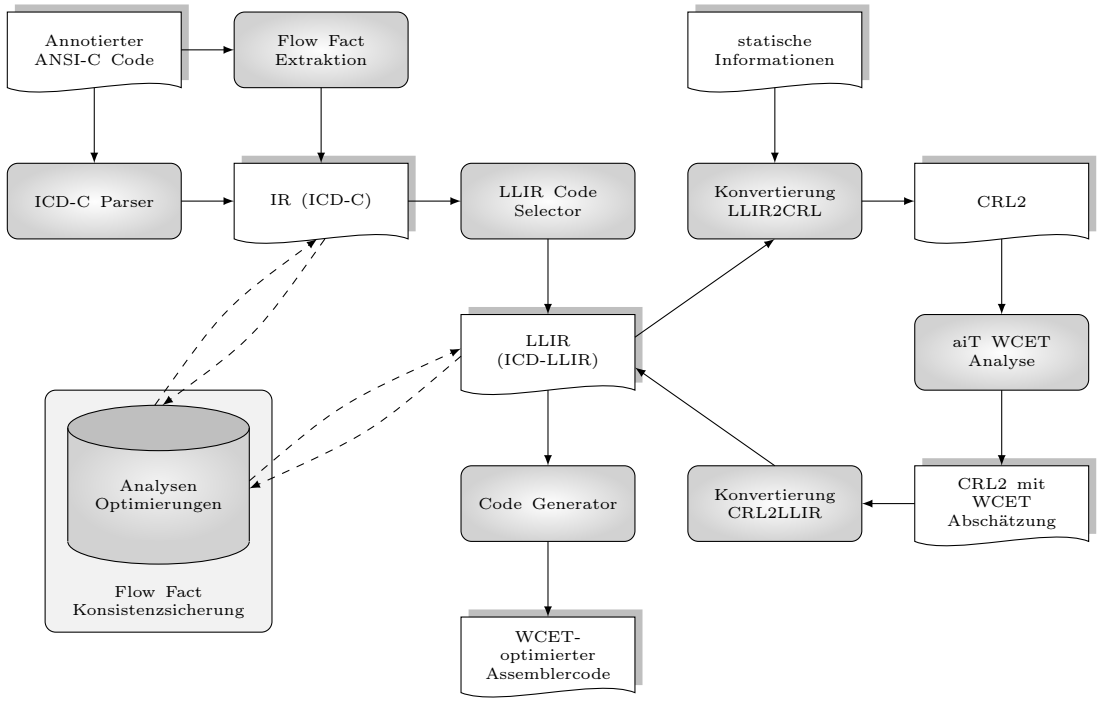


Abbildung 5.1: WCC: Aufbau nach der Diplomarbeit

Die Definition und Modellierung von Flow Facts in Kapitel 4 wiederum hat gezeigt, dass Flow Facts vom Kontrollfluss ihres Quellprogramms abhängig sind und folglich – wenn das Quellprogramm modifiziert wird – u. U. auch transformiert werden müssen, um weiterhin sicher und präzise die Menge möglicher Kontrollflüsse zu beschreiben. Aber nicht nur Änderungen am Kontrollfluss haben Auswirkungen auf Flow Facts.

Beispiel: *Ein Statement* A *– durch eine Flowrestriction referenziert – wird entfernt, ohne dass der Kontrollfluss modifiziert wird. Dennoch ist die Flowrestriction zu aktualisieren, da eine Beschreibung (un)ausführbarer Kontrollflusspfade nicht länger mit Hilfe von Statement* A *möglich ist.*

Jede Modifikation eines Quellprogramms im Compiler ist folglich dahingehend zu untersuchen, ob Flow Facts von dieser Änderung betroffen sein können, und wie ggf. betroffene Flow Facts aktualisiert werden können, um eine konsistente Beschreibung (un)ausführbarer Kontrollflusspfade zu gewährleisten.

Von der Spezifikation der Flow Facts im *annotierten ANSI-C Code* bis zu deren Übergabe an aiT im *CRL2* sind alle Transformationen eines Quellprogramms im WCC potentiell mit Auswirkungen auf Flow Facts verbunden (vgl. Abb. 5.1 auf der vorherigen Seite). Dabei sind zwei Ursachen für Transformationen von Flow Facts unterscheidbar.

Beim Wechsel der Repräsentation eines Quellprogramms vom Quellcode zu einer Intermediate Representation bzw. zwischen zwei Intermediate Representations des WCCs ist auch die Darstellung der Flow Facts für die neue Intermediate Representation anzupassen. Da dieser Schritt primär eine Übersetzung derer Bezugspunkte ist, wird diese Transformation auch als *Translation* bezeichnet. Die Translationen von Flow Facts werden im Abschnitt 5.3 vorgestellt.

Durch Optimierungen auf der IR sowie der LLIR wird der Kontrollfluss eines Quellprogramms z. T. wesentlich verändert. Ein *Update* von Flow Facts kann notwendig sein, um auch trotz dieser Änderungen die Sicherheit und Präzision der Beschreibung (un)ausführbarer Kontrollflusspfade durch Flow Facts weiter zu gewährleisten. Updates von Flow Facts werden ab Abschnitt 5.4 näher untersucht.

Definition 5.1
Wird eine Transformation von Flow Facts durch eine Strukturänderung in einer Programmrepräsentation notwendig, so wird von einem **Update** der Flow Facts gesprochen. Werden dagegen Transformationen von Flow Facts durch einen Wechsel der Programmrepräsentation notwendig, so werden diese **Translation** genannt.

Eine grundlegende Analyse, welche Anforderungen an Transformationen von Flow Facts gelten müssen, um auch nach diesen eine sichere und präzise $\mathrm{WCET}_{\mathrm{est}}$-Berechnung zu ermöglichen, wird in Abschnitt 5.2 durchgeführt. Zuvor werden in Abschnitt 5.1 Transformationen von Flow Facts vorgestellt, wie sie in anderen Projekten realisiert wurden.

5.1 Transformationen in der Literatur

Ausgehend davon, dass annotierte Flow Facts i. d. R. aus der *High Level Intermediate Representation* eines Quellprogramms oder gar aus dessen Quellcode gewonnen werden, sind Transformationen notwendig, um diese – wie in Abb. 3.6 auf Seite 23 gezeigt – für eine auf einer *Low Level Intermediate Representation* basierende WCET-Analyse verfügbar zu machen. Die notwendigen Transformationen lassen sich bezüglich ihrer Methodik in drei Kategorien einordnen:

1. **Transformationen durch den Benutzer**

   Transformationen können durch den Programmierer manuell vorgenommen werden, so dass dieser direkt Flow Facts mit Bezug zum Assemblercode annotiert. Dies ist z. B. der typische Vorgang, um aiT oder Bound-T mit Flow Facts zu versorgen.

Durch manuelle Transformationen lassen sich alle Arten von Optimierungen unterstützen, da der Programmierer die Flow Facts zur Not auf Assemblerebene neu formulieren kann. Jedoch sind Aufwand und Fehleranfälligkeit extrem hoch. Zudem benötigt der Programmierer umfassende Kenntnis der Transformationen des Quellprogramms durch den Compiler.

2. **Compiler parallele Transformationen**

   Transformationen können parallel zu den Strukturänderungen durch einen Compiler stattfinden. Dazu wird z. B. in [Engblom 1997; Engblom u. a. 1998] die *ODL* (*Optimization Description Language*) vorgestellt, die dem Compiler die Beschreibung der Transformationen durch ein *Inpattern* – einer Strukturbeschreibung der betroffenen Elemente vor der Transformation – und ein *Outpattern* – eine Strukturbeschreibung der entsprechenden Elemente nach der Transformation – erlaubt. Diese Daten können dann zur Modifikation der Flow Facts genutzt werden.

   Alle Optimierungstechniken sind um entsprechende Beschreibungsmechanismen zu erweitern, so dass das Wissen, über welches in der manuellen Transformation der Programmierer verfügen muss, hier explizit durch die Optimierungen formuliert wird. Letztendlich ist dieser Ansatz ähnlich mächtig wie die Erweiterung eines Compiler zur integrierten Transformation von Flow Facts, da zum Einen die ODL zur Beschreibung beliebiger Transformationen erweitert werden kann, zum Anderen aber eine automatisierte Auswertung dieser Beschreibungen erfolgt, so dass manuelle Neuformulierungen nicht länger möglich sind.

3. **Erweiterung eines bestehenden Compilers um Transformationen**

   In [Kirner 2003] wird schließlich vorgestellt, wie ein Compiler erweitert werden kann, um auf einem Kontrollflussgraphen basierende Flow Facts zu transformieren. Diese Erweiterung besteht im Wesentlichen aus den drei Bereichen

   - *Markerbezüge* (*Marker Bindings*) korrigieren,
   - *Einschränkungen* (*Restrictions*) aktualisieren sowie
   - *Schleifeninformationen* (*Loop Informations*) anpassen.

   Dazu werden zu einer Reihe von Optimierungen konkrete Regeln vorgestellt, die sich aus den fünf Operationen Einfügen, Verschieben, Kopieren, Löschen und Ersetzen einzelner Flow Fact Elemente zusammensetzen.

   Wenn ein Compiler keinen Kontrollflussgraphen unterstützt, so ist für diesen Ansatz ein Kontrollflussgraph parallel zur eigentlichen Datenstruktur zu halten und zudem eine Funktion notwendig, die dessen Aktualisierung vornimmt.

Entsprechend der Modellierung von Flow Facts in der Literatur mit Bezug auf einen Kontrollflussgraphen basieren auch die automatisierten Transformationen von Flow Facts auf diesem Graphen. Da im Rahmen dieser Arbeit jedoch ein anderer Ansatz verfolgt wird, sind natürlich auch die Transformationen für diesen nicht direkt mit denen in der Literatur vergleichbar.

Allerdings ist, auch wenn die Techniken von Raimund Kirner, wie beschrieben, nicht ohne weiteres adaptierbar sind, prinzipiell dessen Vorgehen aufzugreifen. Da Flow Facts als Bestandteil der Compiler-Datenstrukturen gehalten werden, ist der Compiler um Mechanismen zur Transformation von Flow Facts zu erweitern.

5.2 Sicherheit und Präzision

Ziel der Transformation von Flow Facts ist, durch diese auch weiterhin eine *sichere* und *präzise* Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ zu ermöglichen. Für Translationen, den Übersetzungen von Flow Facts, ist dies kaum problematisch. Für den Entwurf und Einsatz von Updatemechanismen für Flow Facts hingegen ist wichtig zu wissen, wann solch ein Update sicher und präzise bzgl. der $\mathrm{WCET}_{\mathrm{est}}$-Berechnung ist.

Letztendlich lassen sich alle Updates von vorhandenen Flow Facts und insbesondere für Flowrestrictions auf eine Ersetzung von deren Bestandteilen zurückführen. So entspricht z. B. das Löschen eines Summanden in einer Flowrestriction dessen Ersetzung durch Null.

Das Kopieren von Flow Facts, also das wiederholte Einfügen einer Information, gefährdet offensichtlich weder Sicherheit noch Präzision der $\mathrm{WCET}_{\mathrm{est}}$-Berechnung und bedarf keiner weiteren Betrachtung. Das Erzeugen neuer Flow Facts ist kein Update im eigentlichen Sinne. Die Sicherheit und Präzision dieses Schritts ist durch den Programmierer der erzeugenden Klasse zu gewährleisten und ebenfalls nicht Bestandteil dieser Betrachtung.

Im Folgenden wird daher im Abschnitt 5.2.1 untersucht, unter welchen Voraussetzungen eine Ersetzung einer Entscheidungsvariablen in einer Flowrestriction sicher ist. In Abschnitt 5.2.2 werden die Begriffe gleichwertige und abschätzende Ersetzung eingeführt, die für die Updatemechanismen von Flowrestrictions unter Voraussetzung der Sicherheit deren Präzision in den Fokus nehmen.

Abschnließend wird in Abschnitt 5.2.3 die Sicherheit des Austauschs einer Entscheidungsvariablen in einer Loopbound analysiert.

5.2.1 Sicherheit einer Ersetzung in Flowrestrictions

Gegeben sei eine Flowrestriction nach (4.2):

$$a^T \cdot x \leq b^T \cdot x \text{ mit } \forall j \in J \text{ gilt: } a_j, b_j \geq 0 \tag{5.1}$$

Wird eine Entscheidungsvariable x_j in (5.1) durch eine Entscheidungsvariable $x_k \in X$ ersetzt mit

$$x_j = x_k, \tag{5.2}$$

so ist die Sicherheit dieser Ersetzung offensichtlich.

Gibt es keine solche Entscheidungsvariable x_k und eine Ersetzung eines x_j ist dennoch notwendig, so kann das Vorkommen von x_j nur abgeschätzt werden. Sei x_j nun Element der „kleiner gleich Seite“ der Ungleichung, also $a_j > 0$ und $b_j = 0$.

Die Abschätzung von

$$x_j \in X \text{ mit } b_j = 0$$

durch

$$\sum_{k \in \hat{X}} c_k \cdot x_k \text{ mit } \hat{X} \subset X$$

in (5.1) ist sicher, wenn gilt:

$$\sum_{k \in \hat{X}} c_k \cdot x_k \leq x_j. \tag{5.3}$$

Die Abschätzung von (5.1) ist somit:

$$a^T \cdot x - a_j \cdot x_j + a_j \cdot \sum_k c_k \cdot x_k \leq b^T \cdot x. \tag{5.4}$$

Die Ersetzung einer Entscheidungsvariablen der „kleiner gleich Seite“ ist also sicher, wenn diese nach unten abgeschätzt wurde.

Beweis. Sei $\mathbb{L}$ die Menge aller Lösungen für (5.1). Wegen (5.3) gilt:

$$a^T \cdot x - a_j \cdot x_j + a_j \cdot \sum_k c_k \cdot x_k \leq a^T \cdot x \leq b^T \cdot x. \tag{5.5}$$

Daraus folgt aber für alle $l \in \mathbb{L}$, dass l auch eine Lösung der Abschätzung (5.4) ist. Damit ist insbesondere auch die Lösung, die dem WCET-Pfad entspricht, Lösung der Abschätzung. Diese ist somit sicher. □

Analog zu einem Element der „kleiner gleich Seite“ lässt sich zeigen, dass ein Element der „größer gleich Seite“ (x_j mit $b_j > 0$ und $a_j = 0$) für eine sichere Ersetzung nach oben abzuschätzen ist.

5.2.2 Präzision einer Ersetzung in Flowrestrictions

Die Präzision einer Ersetzung für eine Flowrestriction ist schwerer als deren Sicherheit zu fassen, da das intuitive Maß der Differenz

$$\Delta = a_j \cdot x_j - a_j \cdot \sum_k c_k \cdot x_k \tag{5.6}$$

dazu nicht direkt geeignet ist und höchstens als Indiz dienen kann.

Unter anderem durch die Flowrestrictions wird der Lösungsraum des Optimierungsproblems „Bestimmung der maximalen Ausführungszeit eines Programms“ als n-dimensionaler konvexer Polyeder beschrieben, dessen Punkte mit ganzzahligen Koordinaten Lösungen dieses Optimierungsproblems sein können (letztendlich ergänzen die Flow Facts aiTs Beschreibung des Lösungsraums des dort formulierten Optimierungsproblems; wird kein Polyeder beschrieben, so ist das Optimierungsproblem unbeschränkt oder ohne gültige Lösung und aiT bricht mit einer Fehlermeldung ab) [Recht 2004].

Jede Flowrestriction beschreibt dabei einen Halbraum begrenzt durch eine Hyperebene, die so den Lösungsraum mit einschränken kann. Durch die Modifikation der Ungleichung wird die zugehörige Hyperebene verschoben, wobei diese Verschiebung wegen der Sicherheit einer Transformation den Lösungsraum nicht verkleinern, sondern höchstens vergrößern darf.

Liegt eine entsprechende Hyperebene aber z. B. vollständig außerhalb des Lösungsraums, so hat deren Verschiebung keinen Einfluss auf diesen und somit auf das Ergebnis der Analyse. Eine entsprechende Ersetzung wäre auch bei großem Δ präzise bzgl. des Optimierungsproblems.

Auch wenn durch diese Hyperebene der Lösungsraum begrenzt wird, ist eine Ersetzung nicht zwangsläufig unpräzise, denn solange die durch die Verschiebung der Hyperebene in den Polyeder neu aufgenommenen Punkte mit ganzzahligen Koordinaten einen kleineren Zielfunktionswert liefern als die bisher optimale Lösung des Maximierungsproblems, ist keine Verschlechterung der $\mathrm{WCET}_{\mathrm{est}}$-Berechnung auszumachen.

Durch die Ganzzahligkeitsbedingung der Entscheidungsvariablen kann sogar eine Verschiebung einer Hyperebene ohne die Aufnahme eines neuen Punkts in den Lösungsraum enden.

Eine erste Sensibilitätsanalyse über die Auswirkung der Änderung eines einzigen Elements in einer Nebenbedingung eines Optimierungsproblems ist in [Recht 2004] zu finden. Da im WCC bei einer Ersetzung i. d. R. mindestens zwei Elemente geändert werden und darüber hinaus weder die Zielfunktion noch das vollständige Nebenbedingungssystem bekannt sind (beide werden durch aiT erstellt), ist eine genaue Analyse der Präzision einer Ersetzung im Rahmen dieser Arbeit nicht möglich.

Statements mit gleicher Bedeutung

In ICD-C werden Flowrestrictions in Abhängigkeit von Statements verwaltet, d. h. in den Flowrestrictions werden Zeiger auf Statements gehalten, die für das Model in (4.2) natürlich durch deren Entscheidungsvariable für die Ausführungshäufigkeit auf einem Kontrollflusspfad zu ersetzen sind. Aber auch wenn in ICD-C die Flowrestrictions in Abhängigkeit von Statements formuliert werden, werden sie letztendlich nicht für Statements, sondern für Basisblöcke im CRL2 für die $\mathrm{WCET}_{\mathrm{est}}$-Analyse annotiert.

```
   /* Abbildung auf gleiche Ent- */
2  /* scheidungsvariable für aiT */

4 void A{
    ...
6   while( Bed ){
      ExpStmt A1;
8     ExpStmt A2;
    }
10  ...
  }
```

Abbildung 5.2: Gleichwertigkeit wegen gleicher Entscheidungsvariable

```
   /* Abbildung auf versch. Ent-*/
2  /* scheidungsvariablen für aiT */

4 void B{
    ...
6   ExpStmt B1;
    if( Bed )
8     ExpStmt B2;
    ExpStmt B3;
10  ...
  }
```

Abbildung 5.3: Gleichwertigkeit trotz unterschiedlicher Entscheidungsvariablen

Beispiel: *Die Expression Statements* A1 *und* A2 *in Abb. 5.2 auf der vorherigen Seite können beide in Flowrestrictions referenziert werden. Da aber beide Statements offensichtlich zum gleichen Basisblock gehören, werden sie in CRL2 (nach der Abstraktion auf Basisblöcke bei der Translation ICD-C nach ICD-LLIR) durch dieselbe Referenz (und damit auch durch dieselbe Entscheidungsvariable) repräsentiert.*

In einer Flowrestriction der ICD-C sind folglich Referenzen auf A1 *und* A2 *als gleich anzusehen. Wird z. B. Statement* A1 *entfernt, so können alle Referenzen in Flowrestrictions von* A1 *durch welche auf* A2 *ersetzt werden, ohne dass das Entscheidungsmodell von aiT beeinflusst wird.*

Statements, die auf denselben Basisblock von CRL2 abgebildet werden, sind folglich bezüglich der WCET-Analyse nicht zu unterscheiden, da ihnen in letzter Instanz dieselbe Entscheidungsvariable zugeordnet wird.

Statements mit gleicher Funktion

Aber nicht nur Statements, die dem selben Basisblock zugeordnet werden, eigenen sich für die gegenseitige Stellvertreterschaft in Flowrestrictions:

Beispiel: *In Abb. 5.3 auf der vorherigen Seite liegen die Statements* B1 *und* B3 *wegen des sie trennenden if-Statements nicht im selben Basisblock. Dennoch werden die Statements stets gleich oft ausgeführt, und auch für einen Benutzer, der den Code annotiert, sind diese beiden Statements offensichtlich gleichwertig.*

Auch wenn zwei Statements nicht im selben Basisblock liegen, kann es mitunter möglich sein, dennoch beide in Flowrestrictions zu verwenden, als seien sie gleich. Das liegt daran, dass das System an Nebenbedingungen für die Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ primär in aiT aufgestellt wird. Im vorangegangenen Beispiel wird u. a. ermittelt, dass der Basisblock mit B1 und der Basisblock mit B3 stets gleich oft ausgeführt werden. Eine Information à la 1*B1 = 1*B3 wird in die Nebenbedingungen aufgenommen (vgl. auch (5.2), deren Formulierung diesem Sachverhalt entspricht).

Solange aiT einen derartigen Zusammenhang zwischen zwei Basisblöcken ermittelt, können deren Statements in ICD-C, aber auch deren Basisblöcke in der LLIR genutzt werden, als seien sie gleich.

Allerdings kann ohne Kenntnis von aiTs internen Methoden nicht beurteilt werden, inwiefern automatisch diese Verbindungen zwischen Basisblöcken erkannt werden können. Es ist daher eine Entscheidungsregel zu finden, wann auch für aiT zwei offensichtlich stets gleich oft ausgeführte Statements oder Basisblöcke als solche erkennbar gelten sollen und dementsprechend ohne Präzisionsverlust als solche benutzt werden können.

Vorher anzumerken ist jedoch noch, dass – auch wenn eine derartige Verbindung durch aiT nicht erkannt wird – eine gegenseitige Ersetzung zumindest sicher ist. Allerdings kann der Präzisionsverlust so hoch sein, dass die Berechenbarkeit einer $\mathrm{WCET}_{\mathrm{est}}$ nicht länger gewährleistet ist. Daher ist die Entscheidungsregel konservativ (mit hoher Entscheidungssicherheit) zu wählen.

Klassifizierung von Ersetzungen

Definition 5.2
Eine Entscheidungsvariable in einer Flowrestriction kann durch eine gewichtete Summe von Entscheidungsvariablen **gleichwertig ersetzt** werden, wenn

1. sich der Wert der Entscheidungsvariablen und der der gewichteten Summe von Entscheidungsvariablen auf dem wahren WCET-Pfad entsprechen und
2. dieses Entsprechung auch von aiT rekonstruiert werden kann.

Die Modellentwicklung durch aiT basiert auf einer (umfassenden) Kontrollflussanalyse. Entsprechend ist davon auszugehen, dass die Verbindung zwischen Entscheidungsvariablen, die durch eine Kontrollflussanalyse in ICD-C oder der ICD-LLIR (also auf Grundlage von Basisblöcken) als gleichwertig ermittelt wurde, auch in aiT rekonstruiert werden kann. Obwohl die Gleichwertigkeitsanalyse auf Basisblöcken beruht, werden die Flowrestrictions in ICD-C selbstverständlich weiterhin in Abhängigkeit von Statements (in diesen Basisblöcken) formuliert.

Problematischer sind die Schlussfolgerungen auf Gleichwertigkeit, die durch eine Semantikanalyse der ICD-C Statements gewonnen werden, da diese Semantik nicht mehr auf der niedrigen Abstraktionsebene von aiT verfügbar ist. Daher wird die Möglichkeit für eine gleichwertige Ersetzung im Folgenden auf Statements aus einer Schleife eingeschränkt, um auf hohem Sicherheitsniveau die Abhängigkeitsrekonstruktion zu gewährleisten.

Bei einer manuellen Ersetzung kann auf diese Bedingung verzichtet werden, wenn der Programmierer einer Optimierung anderweitig sicher ist, dass die Abhängigkeitsrekonstruktion weiterhin gelingt, z. B. in dem er entsprechende Flow Facts explizit hinzufügt.

Kann für eine Entscheidungsvariable keine gleichwertige Ersetzung gefunden werden, so kann eine Abschätzung durchgeführt werden. Dafür ist die Definition 5.2 wie folgt zu modifizieren:

Definition 5.3
Eine Entscheidungsvariable in einer Flowrestriction kann durch eine gewichtete Summe von Entscheidungsvariablen **abschätzend ersetzt** werden, wenn

1. das Verhältnis des Werts der Entscheidungsvariablen und der gewichteten Summe von Entscheidungsvariablen auf dem wahren WCET-Pfad bekannt ist („größer gleich“ oder „kleiner gleich“) und
2. dieses Verhältnis auch von aiT rekonstruiert werden kann.

Mit der Kenntnis des Verhältnisses kann entschieden werden, ob durch eine abschätzende Ersetzung eine Abschätzung nach oben oder eine Abschätzung nach unten für eine Flowrestriction möglich ist. Für die Rekonstruktion des Verhältnisses werden die gleichen Annahmen wie zur Rekonstruktion einer Entsprechung zugrunde gelegt.

5.2.3 Sicherheit einer Ersetzung in Loopbounds

Gegeben ist eine Loopbound nach Modell (4.9):

$$min_{annotiert} \leq \tilde{x}_l \leq max_{annotiert} \tag{5.7}$$

Die Entscheidungsvariable $\tilde{x}_l$ in (5.7) kann durch eine Entscheidungsvariable $\tilde{x}_n \in \tilde{X}$ sicher ersetzt werden, wenn

$$\text{a)} \quad min_{annotiert} \leq \tilde{x}_n \leq max_{annotiert} \quad \text{oder} \tag{5.8}$$

$$\text{b)} \quad min_l \leq min_n \ \text{ und } \ max_l \geq max_n \tag{5.9}$$

gilt.

Während (5.8) offensichtlich für eine sichere Ersetzung reicht, ist dies für (5.9) zu zeigen:

Beweis. Die Ersetzung einer Entscheidungsvariablen durch die Entscheidungsvariable $\tilde{x}_n$ ist sicher, wenn $min_{annotiert} \leq \tilde{x}_n \leq max_{annotiert}$ gilt. Nach (5.7) gilt:

$$min_{annotiert} \leq \tilde{x}_l \leq max_{annotiert} \tag{5.10}$$

Mit (5.9) folgt unmittelbar:

$$min_{annotiert} \leq min_l \leq \tilde{x}_n \leq max_l \leq max_{annotiert} \tag{5.11}$$

□

Für eine Ersetzung einer Schleife in einer Loopbound muss nicht zwingend die genaue Iterationshäufigkeit einer neuen Schleifen bekannt sein, sondern es genügt zu wissen, dass die neue Schleife mindestens so oft aber auch höchstens so oft wie die zu ersetzende Schleife iteriert wird.

Analog ist offensichtlich richtig, in einer annotierten Loopbound $min_{annotiert}$ nach unten und $max_{annotiert}$ nach oben abzuschätzen. Dies kann notwendig werden, wenn $min_{annotiert}$ oder $max_{annotiert}$ nicht ganzzahlig sind.

Beispiel: *Wird die Iterationshäufigkeit einer Schleife in Folge der ICD-C Optimierung Loop Unrolling um Faktor u reduziert, so sind in einer Loopbound die annotierte minimale Ausführungshäufigkeit min durch* $\lfloor \frac{min}{u} \rfloor$ *und die annotierte maximale Ausführungshäufigkeit max durch* $\lceil \frac{max}{u} \rceil$ *zu ersetzen, um weiterhin eine sichere aber auch möglichst präzise Annotation zu gewährleisten.*

5.3 Translation von Flow Facts

Zur Translation von Flow Facts werden die drei Klassen

- FlowfactmanagerCtoICDC,
- FlowfactmanagerICDCtoLLIR und
- FlowfactmanagerLLIRtoCRL

eingesetzt. Deren Arbeit erfolgt im Wesentlichen in zwei Schritten:

1. Anlegen eines *Wörterbuchs*

   Die Abbildung von Elementen der ursprünglichen Abstraktionsebene in jene der neuen Abstraktionsebene wird durch eine Art Wörterbuch beschrieben. Dieses wird i. d. R. während der Übersetzung der Repräsentation eines Quellprogramms angelegt.

2. Translation der Flow Facts in die *Zielsprache*

   Nach der Übersetzung der Repräsentation des Quellprogramms in eine neue Intermediate Representation werden die Flow Facts mittels des angelegten Wörterbuchs übersetzt.

Diese beiden Schritte werden in den folgenden Unterabschnitten für die notwendigen Translationen näher erläutert.

5.3.1 Translation C nach ICD-C

Gegeben ist eine ICD-C Datenstruktur für ein Quellprogramm $\mathcal{P}$. Die Marker und Flow Facts aus dem Quellcode wurden durch den ICD-C Parser in ihrer Form unverändert als Pragmas den ICD-C Elementen hinzugefügt.

Für die Extraktion von Flow Facts aus diesen Pragmas wird eine Abbildung von den Markern, mit denen der Programmierer die Anweisungen im Quellprogramm $\mathcal{P}$ gekennzeichnet hat, auf die entsprechenden Statements von ICD-C benötigt.

$$\textit{Wörterbuch C nach ICD-C}: \ \textit{Marker} \rightarrow \textit{IR\_Stmt} \tag{5.12}$$

Für jedes Pragma vom Typ Marker wird der Marker mit dem Statement, dem dieses Pragma zugeordnet wurde, im Wörterbuch eingetragen. Wird das Pragma von einer Expression gehalten, so ist die Übersetzung deren Expression Statement. Marker für Compound Statements oder *for*-Statements ohne Abbruchbedingung werden nicht aufgenommen, da diese nicht durch Flow Facts annotiert werden dürfen (vgl. Abschnitt 4.5.1). Marker auf Symboldeklarationen und Typbezeichner (IR_Symbol-Objekte bzw. IR_Type-Objekte sind neben den Statements und Expressions mögliche Ziele für Pragmas) werden verworfen, da sie keine annotierbaren Objekte darstellen.

Zusätzlich wird jede Funktion in ICD-C mit dem Funktionsnamen als Marker und dessen Top Compound Statement als Übersetzung im Wörterbuch eingetragen. Mehrfach definierte Marker werden verworfen (entsprechende Warnungen verstehen sich von selbst).

Mithilfe des Wörterbuchs werden abschließend alle Flow Facts des Quellprogramms übersetzt. Dabei gilt:

- Loopbounds werden nur übersetzt, wenn das Ziel ein Loop Statement ist und dieses im Falle eines *for*-Statements eine Abbruchbedingung enthält, sonst verworfen.
- Flowrestrictions werden nur übersetzt, wenn alle Marker im Wörterbuch stehen, sonst verworfen.
- Flowrestrictions werden dabei automatisch in die Form nach (4.2) gebracht.
- Flowrestrictions ohne Aussage ($0 \leq 0$ sowie $0 \leq SUM$) werden verworfen.

Ergebnis ist eine Flow Fact annotierte ICD-C Repräsentation des Quellprogramms $\mathcal{P}$.

5.3.2 Translation ICD-C nach ICD-LLIR

Gegeben ist eine Flow Fact annotierte, ggf. optimierte ICD-C Datenstruktur eines Quellprogramms $\mathcal{P}$. Für jede Compilation Unit der IR wird eine eigene LLIR erzeugt. Dementsprechend wird die Übersetzung der Flow Facts auch für jede Compilation Unit einzeln angestoßen.

Die Übersetzung der Flow Facts von ICD-C in die ICD-LLIR wird genutzt, um von Statements zu abstrahieren und Flow Facts auf Ebene der Basisblöcke zu formulieren. Dementsprechend bildet das Wörterbuch Statements der IR auf Basisblöcke der LLIR ab.

$$\textit{Wörterbuch ICD-C nach ICD-LLIR}: \ \textit{IR\_Stmt} \rightarrow \textit{LLIR\_BB} \tag{5.13}$$

Das Wörterbuch wird während der Code Selection durch den LLIR Code Selector gefüllt. Dieser erzeugt nach einer Kontrollflussanalyse auf der IR für jeden derer Basisblöcke mindestens einen Basisblock in der LLIR. Dieser wird Ziel aller Referenzen von Flow Facts auf Statements des ursprünglichen Basisblocks der IR. Die Top Compound Statements der Funktionen der IR, denen kein Basisblock direkt gegenübersteht, werden auf den jeweils ersten Basisblock der entsprechenden LLIR Funktion abgebildet.

Durch diese Abbildung werden den *while*- und *do-while*-Statements die Basisblöcke ihrer Abbruchbedingung zugewiesen. Analog dazu werden *for*-Statements, die selbst nicht Teil eines Basisblockes in der IR sind, auf den LLIR Basisblock ihrer Abbruchbedingung abgebildet. Referenzen auf *for*-Statements ohne Abbruchbedingung sind nicht erlaubt (vgl. Abschnitt 4.5.1).

Mithilfe des Wörterbuchs werden dann alle Flow Facts der IR übersetzt. Dabei gilt:

- Flowrestrictions und Loopbounds werden nur übersetzt, wenn sie vollständig Teil der aktuellen Compilation Unit bzw. LLIR sind, sonst verworfen. Dies entfällt, sobald eine gemeinsame Analyse aller LLIRs durch aiT ermöglicht wird (vgl. Abschnitt 4.5.2).
- Da das Wörterbuch keine surjektive Abbildung von Statements der IR auf Basisblöcke der LLIR bietet, weil mehrere Statements der IR durchaus dem gleichen LLIR Basisblock zugeordnet werden können, können Flowrestrictions u. U. vereinfacht werden. Dies erfolgt automatisch, um weiterhin die Form nach (4.2) zu erhalten.
- Flowrestrictions ohne Aussage ($0 \leq 0$ sowie $0 \leq SUM$) werden verworfen.

Drei Besonderheiten sind zu berichten:

```
    /* Head Controlled Loop */
2   ...
  _Pragma("loopbound min 2 max 5")
4 for( i=0; i<5; i++ ){
    Anweisungsblock A;
6 }
  ...
```

Die Bedingung wird 6 mal getestet.
Annotiere für aiT min=3 max=6.

```
    /* Tail Controlled Loop */
2   ...
  _Pragma("loopbound min 2 max 5")
4 do{
    Anweisungsblock A;
6 }while( i=0; i<5; i++ );
  ...
```

Die Bedingung wird 5 mal getestet.
Annotiere für aiT min=2 max=5.

Abbildung 5.4: Bsp. Testhäufigkeit einer Schleifenbedingung

- Auch für nicht annotierte Schleifen (per *for*- (mit Abbruchbedingung), *while*- oder *do-while*-Statement definiert) wird eine Loopbound mit der Iterationshäufigkeit $min = max = -1$ erzeugt. Dadurch sind alle Schleifen von ICD-C auch noch in der ICD-LLIR an der annotierten Abbruchbedingung zu erkennen. Dies kann für eine Analyse in der LLIR hilfreich sein.
- Durch die Code Selection werden neue Schleifen erzeugt (z. B. zur Initialisierung des Speicherplatzes einer Union). Diese werden automatisch durch Loopbounds mit ihrer genauen Iterationshäufigkeit annotiert.
- Schleifen werden in aiT nicht mit ihrer Iterationshäufigkeit sondern mit der Anzahl der Tests der Schleifenabbruchbedingung annotiert. Da in der LLIR nicht bekannt ist, ob eine Abbruchbedingung zum Beginn oder zum Abschluss einer Iteration geprüft wird, werden die Loopbounds schon bei der Übersetzung in die LLIR für aiT angepasst. Das ist gleichbedeutend damit, dass *min* und *max* von *for*- und *while*-Statements um eins erhöht werden (vgl. Abb. 5.4). Diese automatische Anpassung erlaubt, dass der Programmierer im Quellcode für alle Schleifen einheitlich und intuitiv Iterationshäufigkeiten annotieren kann.

Ergebnis ist pro Compilation Unit jeweils eine Flow Fact annotierte ICD-LLIR Datenstruktur.

5.3.3 Translation ICD-LLIR nach CRL2

Gegeben ist eine Flow Fact annotierte, ggf. optimierte ICD-LLIR Datenstruktur eines Quellprogramms $\mathcal{P}$.

Das Wörterbuch zur Translation der Flow Facts nach CRL2 wird während der Konvertierung der LLIR in die CRL2 Datenstruktur durch den Konverter LLIR2CRL gefüllt. Da die Strukturen von LLIR und CRL2 sehr ähnlich sind (beide Intermediate Representations sind Assembler nah), wird jeder LLIR Basisblock auf einen CRL2 Basisblock abgebildet.

$$\textit{Wörterbuch ICD-LLIR nach CRL2}: \ \textit{LLIR\_BB} \rightarrow \textit{CrlItem} \tag{5.14}$$

Mit Hilfe des Wörterbuchs werden alle Flow Facts der LLIR übersetzt. Dabei gilt:

- Flow Facts ohne Aussage ($0 \leq 0$ sowie $0 \leq SUM$) werden verworfen.
- Für markierte Schleifen ohne grundlegende Begrenzungsinformation (max = min = -1) wird eine Warnung ausgegeben.

Ergebnis der Translation ist eine Flow Fact annotierte CRL2 Datenstruktur.

Loopbounds werden mit aiTs Mechanismus zum Annotieren von Schleifen in CRL2 spezifiziert, da dieser Mechanismus eine genauere Analyse als der mit dessen Flow Constraints erlaubt (vgl. Abschnitt 4.5). Allerdings kann dieser Mechanismus nur für Schleifen genutzt werden, die aiT in seiner Schleifentransformation als solche erkennt. Daher müssen z. B. Multi-Entry Loops durch den Programmierer durch Flowrestrictions statt Loopbounds annotiert werden.

Wünschenswert wäre hier ein Einlesen der CRL2-Datenstruktur nach der Schleifentransformation, um dem Programmierer eine Rückmeldung zu geben, ob all seine Loopbounds erfolgreich annotiert wurden oder ggf. zu melden, welche Loopbound durch eine Flowrestriction zu ersetzen ist. Die Fehlermeldungen von aiT diesbezüglich sind als kryptisch zu bezeichnen.

5.4 Updates für Flow Facts

Die Auswirkungen einer Optimierung auf seine Intermediate Representation unterscheiden sich bei den verschiedenen Optimierungen im WCC wesentlich. Während einige Optimierungen den Kontrollfluss eines Programms nicht verändern, wie z. B. die *Value Propagation* oder die *Common Subexpression Elimination*, modifizieren andere Optimierungen den Kontrollfluss wesentlich, insbesondere z. B. Schleifenoptimierungen wie *Loop Unrolling* oder *Loop Collapsing*. Die notwendigen Updates für Flow Facts hängen daher stets von der Optimierung, aber zudem auch vom konkreten Einsatz einer Optimierung ab.

Wegen des engen Zusammenhangs von Optimierung, Einsatz und resultierendem Update von Flow Facts ist es sinnvoll, nur allgemeine Updatemechanismen zu entwickeln, die in den Optimierungen unterstützend genutzt werden können, um so individuell optimale Aktualisierungen der Flow Facts bei geringem Programmieraufwand zu ermöglichen. Der Verzicht auf stark spezialisierte Updatemechanismen für genau eine Optimierung erhöht die Wiederverwendbarkeit der Updatemechanismen und erlaubt die Entwicklung neuer Optimierungen, ohne neue Updatemechanismen entwerfen zu müssen.

Jede Optimierung ist folglich auf ihre Änderungen der Intermediate Representation und die damit notwendigen Aktualisierungen von Flow Facts hin zu untersuchen, entsprechende Updatemechanismen sind innerhalb dieser Optimierung anzustoßen.

Die Klassen IR_Flowfactupdater bzw. LLIR_Flowfactupdater stellen diese Updatemechanismen für ICD-C und ICD-LLIR bereit. Die enge Verflechtung von Optimierung und Flow Fact Update spiegelt sich auch in der Implementierung wieder. Eine Optimierung ererbt die Updatemechanismen von der entsprechenden Flowfactupdater-Klasse.

Die Updatemechanismen lassen sich – wie Abb. 5.5 auf der nächsten Seite zeigt – in mehrere Kategorien einteilen. Es sind Mechanismen speziell für Loopbounds sowie speziell für Flowrestrictions notwendig, um deren unterschiedlicher Semantik und Syntax gerecht zu werden.

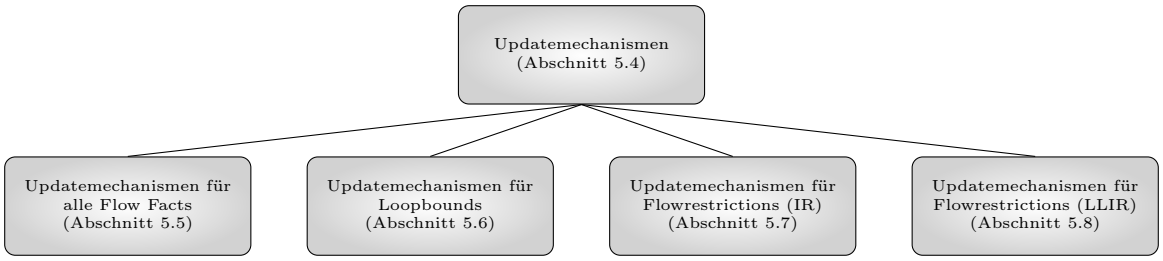


Abbildung 5.5: Updatemechanismen Überblick

Diese könnten zwar auch kombiniert in gemeinsamen Updatemechanismen genutzt werden, dadurch würde allerdings die Flexibilität durch die freie Kombination der Mechanismen und damit die Wiederverwendbarkeit eingeschränkt werden. Letztendlich sind nur zwei gemeinsame Techniken für alle Flow Facts entstanden, die in Abschnitt 5.5 vorgestellt werden.

Wie in Abschnitt 5.6 erläutert wird, sind die Updatemechanismen für Loopbounds von geringer Komplexität, da Loopbounds nur begrenzte Ausdruckskraft haben. Zudem ist ihr Einsatz in ICD-C nur für Loop Statements erlaubt, und folglich sind Loopbounds nur von einem Teil der Optimierungen betroffen.

Die Updatemechanismen sind für ICD-C und die ICD-LLIR sehr ähnlich und werden daher gemeinsam vorgestellt, auch wenn in der LLIR praktisch nie Updates für Loopbounds zum Einsatz kommen. Denn ihr Einsatz ist z. Z. nur in der ICD-LLIR Optimierung *Empty Basicblock Elimination* notwendig. Solange aber die Bindung von Loopbound und Abbruchbedingung aus der Translation von ICD-C nach ICD-LLIR bestehen bleibt, wird dort kein Update von Loopbounds notwendig werden.

Genau gegenteilig verhält es sich bei Flowrestrictions. In ihr können nahezu beliebige Elemente zur Beschreibung des möglichen Kontrollflusses genutzt werden. Sie sind von den meisten Optimierungen betroffen und wegen ihrer universellen Aussagekraft schwerer zu aktualisieren. Zudem unterscheiden sich die Techniken für ICD-C und ICD-LLIR wesentlich. Während in ICD-C die Optimierungen und somit zwangsläufig auch die Updatemechanismen stark von der Semantik der Statements geprägt werden, sind in der ICD-LLIR die für Flow Facts relevanten Optimierungen insbesondere von der Struktur des Kontrollflussgraphen abhängig. Daher werden die Techniken getrennt in den Abschnitten 5.7 und 5.8 vorgestellt.

Punktuell wird gezeigt werden, wie diese Updatetechniken in Optimierungen zum Einsatz gelangen können. Dabei wird die Betrachtung auf die für die Updatemechanismen wesentlichen Aspekte beschränkt. Eine ausführliche Analyse aller Optimierungstechniken im WCC sprengt den Rahmen dieser Arbeit. Jedoch zeigen die Ergebnisse in Kapitel 6, dass die vorhandenen Techniken erfolgreich durch die Updatemechanismen unterstützt werden konnten. Einen Überblick der durch Optimierungen genutzten Mechanismen gewährt Anhang A auf Seite 113.

5.5 Updatemechanismen für alle Flow Facts

Zwei Updatemechanismen, die sowohl Loopbounds als auch Flowrestrictions transformieren, werden in diesem Abschnitt eingeführt.

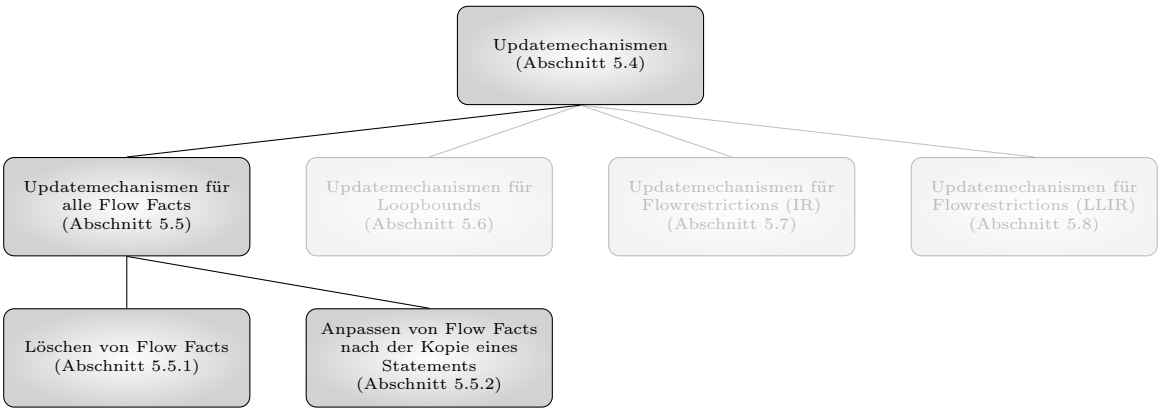


Abbildung 5.6: Updatemechanismen für alle Flow Facts

Zum Einen werden in Abschnitt 5.5.1 Mechanismen vorgestellt, mit denen Flow Facts bzw. Bestandteile von Flow Facts gelöscht werden können. Zum Anderen wird in Abschnitt 5.5.2 ein Mechanismus eingeführt, der in ICD-C nach einer Kopie von Statements mittels derer copy()-Methode Flow Facts aktualisiert.

In Abschnitt 5.5.3 werden abschließend Einsatzszenarien für diese Mechanismen in den Optimierungen präsentiert.

5.5.1 Löschen von Flow Facts

Der Updatemechanismus *Löschen von Flow Facts* erlaubt es, alle Loopbounds für ein Element, alle Flowrestrictions für ein Element oder alle Referenzen von Flow Facts auf ein Element der IR/LLIR zu löschen. Die Unterschiede der Techniken werden im Folgenden genauer betrachtet:

- Löschen aller Loopbounds für ein Element:
  - IR: flowfactsDeleteLoopbound(IR_LoopStmt loop);
  - LLIR: flowfactsDeleteLoopbound(LLIR_BB bb);

  Werden für eine Schleife Loopbounds spezifiziert – i. d. R. höchstens eine Loopbound pro Schleife –, so werden diese durch die flowfactsDeleteLoopbound()-Mechanismen entfernt. Flowrestrictions sind durch diese Updates nicht betroffen.
- Löschen aller Flowrestrictions für ein Element:
  - IR: flowfactsDeleteFlowrestriction(IR_Stmt stmt);
  - LLIR: flowfactsDeleteFlowrestriction(LLIR_BB bb);

Analog zum Löschen aller Loopbounds werden durch diese Methoden alle Flowrestrictions, die ein gegebenes Statement bzw. einen gegebenen Basisblock referenzieren, vollständig entfernt. Loopbounds sind durch diese Updates nicht betroffen.

- Löschen aller Referenzen von Flow Facts auf ein Element:
  - IR: flowfactsRemoveUnreachableStmt(IR_Stmt stmt);
  - IR: flowfactsRemoveUnreachableStmt(set<IR_Stmt> stmts);
  - IR: flowfactsRemoveUnreachableStmt(IR_Function f);
  - LLIR: flowfactsRemoveUnreachableBB(LLIR_BB bb);

  Es werden nun nicht alle Loopbounds und alle Flowrestrictions für die gegebenen Elemente entfernt (dies kann durch Kombination der zuvor vorgestellten Mechanismen erreicht werden), sondern es werden alle Referenzen auf diese Elemente entfernt. Da Loopbounds für genau ein Element spezifiziert werden, werden sie zwar vollständig entfernt, aber bei Flowrestrictions werden nur Summanden mit Referenzen auf diese Elemente entfernt. Dies entspricht einer Ersetzung der Summanden (und damit natürlich letztendlich derer Entscheidungsvariablen) durch Null.

  Durch diese Mechanismen sind in der ICD-C aber nicht nur Loopbounds und Flowrestrictions der spezifizierten Elemente betroffen, das selbe Update wird auch für jeden Nachfahren in der Statement Hierarchie durchgeführt.

Fazit

Das Löschen von Loopbounds und das Löschen von Flowrestrictions für ein Element kann z. B. notwendig sein, wenn kein adäquates Update möglich ist. Durch das Entfernen der entsprechenden Flow Facts wird dann zumindest sichergestellt, dass die Kontrollflusspfadmenge nicht fälschlicherweise zu stark eingeschränkt wird. Somit ist die Sicherheit der Beschreibung durch Flow Facts weiterhin garantiert.

Da aber durch diese Mechanismen grundlegende Begrenzungsinformationen entfernt werden können, kann u. U. die Berechenbarkeit der $\mathrm{WCET}_{\mathrm{est}}$ nicht länger gewährleistet sein (maximal unpräzise), so dass diese Mechanismen vorsichtig einzusetzen sind, z. B. nur, wenn durch neue oder andere bereits vorhandene Flow Facts die Berechenbarkeit/Präzision gewährt bleibt.

Das Löschen aller Referenzen auf Elemente (und auf all deren Nachfahren) bietet ein sicheres und präzises Update für Elemente der Datenstruktur, die z. B. nie ausgeführt und von einer Optimierung aus der Intermediate Representation entfernt werden. Dagegen ist dieses Update nicht sicher, wenn die entsprechenden Elemente ausgeführt werden können und in einer Flowrestriction auf der „größer gleich Seite“ referenziert werden. Dieser Fall ist beim Einsatz des Mechanismus in einer Optimierung auszuschließen.

5.5.2 Anpassen von Flow Facts nach der Kopie eines Statements

Durch die copy()-Methode in ICD-C wird ein Statement inklusive aller Nachfahren in der Statement Hierarchie kopiert. An diese Statements als User Data angebunden sind die Flowfactrefs für einen effizienten Zugriff auf die das Statement referenzierenden Flow Facts, wie

in Abschnitt 4.6 dargestellt wurde. Zur Zeit werden bei der Kopie von Statements aber nicht diese User Data sondern nur die Zeiger auf diese User Data kopiert.

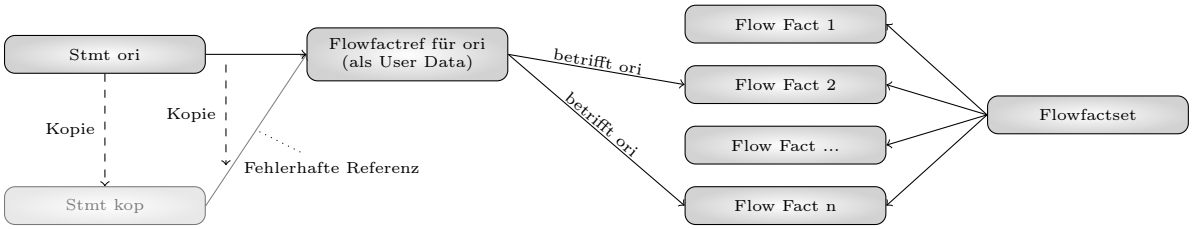


Abbildung 5.7: Problemstellung nach einer Kopie eines Statements mit Flow Facts

Eine copy()-Methode für die Flowfactrefs ließe sich prinzipiell so gestalten, dass statt der Kopie eines Flowfactrefs ein neues, leeres Flowfactref angelegt würde. Aber wegen der Kopie der Zeiger auf User Datas können z. Z. von einer Kopie eines Statements aus über diesen kopierten Zeiger Flow Facts erreicht werden, die keine Referenz auf diese Kopie des Statements halten (vgl. Abb. 5.7). Diese falschen Zeiger auf Flowfactrefs sind selbstverständlich zu entfernen.

```
/* Code vor Optimierung */
2
  ...
4 _Pragma("marker before")
  ExpStmt A;
6
  for( i = 0; i < 2*n, i++ ){
8   _Pragma("marker in")
    ExpStmt B;
10 }

12 _Pragma("flowrestriction 1*in <= 50*before")
14 ...
```

Loop Unrolling um Faktor 2

```
/* Code nach Optimierung */
2
  ...
4 _Pragma("marker before")
  ExpStmt A;
6
  for( i = 0; i < n, i++ ){
8   // Original
    _Pragma("marker in")
10  ExpStmt B;
    // Kopie
12  _Pragma("marker in_kop")
    ExpStmt B;
14 }

16 _Pragma("flowrestriction 1*in + 1*in_kop <= 50*before")
18 ...
```

Abbildung 5.8: Bsp. Kopie von Statements durch *Loop Unrolling*

Abgesehen davon sind nach der Kopie von Statements in der Datenstruktur Flow Facts zu aktualisieren. Dabei sind drei Szenarien denkbar:

- Original-Statement hält eine Loopbound

  Da die Schleifeninformationen für jede Ausführung der Schleife gelten müssen, kann für jede Kopie einer Schleife eine Loopbound mit denselben Informationen angelegt werden.

- Original-Statement hält eine Flowrestriction, deren Bestandteile alle zu diesem Statement und dessen Nachfahren in der Statement Hierarchie gehören

  Dieser Fall ist ähnlich zum Vorausgehenden. Die Flowrestriction muss auch in der Kopie des Statements erfüllt sein. Daher kann eine neue entsprechende Flowrestriction für die Kopie des Statements und dessen Nachfahren erzeugt werden.

- Original-Statement hält eine Flowrestriction, mindestens ein Bestandteil gehört nicht zu diesem Statement und dessen Nachfahren in der Statement Hierarchie

  Original und Kopie zusammen müssen die Forderung bezüglich der nicht zum Statement und dessen Nachfahren gehörenden Bestandteile der Flowrestriction erfüllen. Daher wird für jeden Summanden mit Referenz auf ein Statement, das kopiert wurde, ein Summand mit demselben Faktor für dessen Kopie hinzugefügt.

  Beispiel: *Gegeben ist eine Flowrestriction (Abb. 5.8 auf der vorherigen Seite) mit Referenzen auf Statements, die nicht kopiert werden (*before*), und auf Statements, die kopiert werden (*in*). Um die Aussage der Flowrestriction zu erhalten, ist für jede Kopie eines referenzierten Statements (*in_kop*) ein Summand in der entsprechenden Flowrestriction zu ergänzen. Die Flowrestriction* 1*in + 1*in_kop <= 50*before *in Abb. 5.8 beschränkt auch weiterhin die Ausführung von Statement* B *in Summe auf fünfzig pro Ausführung der gesamten Schleife.*

Das Entfernen falscher Referenzen sowie die Aktualisierung von Flow Facts werden durch die Methode

- IR: flowfactsCorrectCopyOfStmt(IR_Stmt ori, IR_Stmt kop);

– wie in Abb. 5.9 auf der nächsten Seite beschrieben – ausgeführt. Dabei sei cp eine Funktion, die in beliebigen Strukturen eine Referenz auf ein kopiertes Statement durch eine Referenz auf dessen Kopie ersetzt.

Fazit

Dieser Updatemechanismus bietet ein sicheres und präzises Update für Flow Facts, nachdem Statements von ICD-C kopiert wurden.

5.5.3 Einsatzbeispiele

Beispiele für den Einsatz der Updatetechniken *Löschen von Flow Facts* und *Anpassen von Flow Facts nach der Kopie eines Statements* werden im Folgenden vorgestellt.

Löschen von Flow Facts

Da die Entscheidungsvariable in Flowrestrictions durch Null ersetzt und Loopbounds vollständig gelöscht werden, eignet sich das Löschen aller Referenzen auf ein Statement vor allem für Optimierungen wie die *Dead Code Elimination*.

```
flowfactsCorrectCopyOfStmt( IR_Stmt ori, IR_Stmt kop )
begin

  /* Sammle betroffene Flow Facts und entferne falsche Zeiger. */
  Menge<IR_Loopbound> lbs;
  Menge<IR_Flowrestiction> frs;

  for s in kop ∪ Nachfahren( kop ) do
  begin
    lbs += hole_Loopbounds( s );
    frs += hole_Flowrestrictions( s );
    lösche_falschen_Flowfactref_Zeiger( s );
  end

  /* Aktualisiere Loopbounds. */
  for lb in lbs do
    erzeuge_neue_Loopbound( cp( lb ) );

  /* Aktualisiere Flowrestrictions. */
  for fr in frs do
    if( hat_nur_Referenzen_auf( fr, ori ∪ Nachfahren( ori ) ) ) then
      erzeuge_neue_Flowrestiction( cp( fr ) );
    else
    begin
      // Faktor und Referenz
      Paar<int, IR_Stmt> Summand;
      for Summand in hole_Summanden_auf( fr, ori ∪ Nachfahren( ori ) ) do
        if( ist_Summand_der_leq_Seite( fr, Summand ) ) then
          addiere_zu_leq( fr, Summand.first, cp( Summand.second ) );
        else
          addiere_zu_geq( fr, Summand.first, cp( Summand.second ) );
    end

end
```

Abbildung 5.9: Algorithmus: Anpassen von Flow Facts nach der Kopie eines Statements

Beispiel: *Durch die ICD-C Optimierung Dead Code Elimination werden u. a. Statements, die nicht durch den Kontrollfluss erreicht werden können, aus der IR entfernt. Alle Referenzen von Flow Facts auf diese Statements sind vor deren Entfernen aus der IR zu ersetzen.*

Da diese Statements nie ausgeführt werden, ist eine Ersetzung ihrer Ausführungshäufigkeit in Flowrestrictions durch Null sicher und präzise, weswegen ein Update durch den Aufruf von flowfactsRemoveUnreachableStmt(...); *die Flowrestrictions optimal aktualisiert. Zugleich werden alle (nicht länger benötigten) Loopbounds auf diese Statements entfernt.*

Beispiel: *Durch die ICD-C Optimierung Remove Unused Symbols werden Funktionen, die nie aufgerufen werden, vollständig entfernt. Da kein Statement innerhalb dieser Funktion je ausgeführt wird (ein Sprung per goto-Statement ist nur innerhalb einer Funktion erlaubt), ist ein Update der Flow Facts analog zur eben vorgestellten Dead Code Elimination zu erreichen.*

Anpassen von Flow Facts nach der Kopie eines Statements

Der Updatemechanismus ist nach dem Einfügen von Kopien von Statements in ICD-C notwendig.

Beispiel: *Durch die ICD-C Optimierung Loop Unrolling wird die Iterationshäufigkeit einer Schleife um einen Faktor $u > 1$ reduziert. Dafür werden für die Statements des Schleifenkörpers $u - 1$ zusätzliche Kopien in diesen eingefügt (vgl. Abb. 5.8 auf Seite 61). Das Update der Flow Facts erfolgt u. a. durch den Aufruf von* flowfactsCorrectCopyOfStmt(...); *für die kopierten Statements und jeder ihrer Kopien.*

5.6 Updatemechanismen für Loopbounds

Der Konstruktion von Updatemechanismen für Loopbounds kommt zugute, dass diese nur von genau einem Element in ICD-C bzw. in der ICD-LLIR abhängig sind. Darüber hinaus sind auch über dieses Element zusätzliche Informationen bekannt:

- In der IR ist das Statement, auf das sich eine Loopbound bezieht, ein Loop Statement, also entweder ein *for*-Statement (mit Abbruchbedingung), ein *while*-Statement oder ein *do-while*-Statement.
- In der LLIR existiert zwar keine spezielle Struktur für Schleifen, allerdings wird bei der Translation von Loopbounds in die LLIR sichergestellt, dass eine Loopbound stets dem Basisblock mit der Abbruchbedingung der Schleife und somit auch mit einem Sprung zugeordnet wird. Auch bei den im Code Selector erzeugten Schleifen ist stets der Sprungbefehl Teil des Basisblockes mit der Loopbound.

Zudem sind Loopbounds vor allem durch Schleifenoptimierungen in ICD-C betroffen. Diese werden jedoch i. d. R. nur durchgeführt, wenn zusätzliche Informationen über eine Schleife verfügbar sind, z. B. wenn ihre genaue Iterationshäufigkeit ermittelt werden konnte. In der ICD-LLIR sind Schleifen praktisch nicht von Optimierungen betroffen, da insbesondere der Sprungbefehl im annotierten Basisblock durch Techniken wie z. B. der *Loop Invariant Code Motion* nicht modifiziert wird.

Sollten künftige Optimierungen allerdings auch den Sprungbefehl betreffen und diesen nicht nur z. B. durch einen anderen u. U. effizienteren Sprungbefehl im gleichen Basisblock ersetzen (wie z. B. die *Peephole Optimization*), so ist ein neuer Updatemechanismus zu entwickeln, der die Beziehung von annotierter Loopbound und Basisblock mit Sprungbefehl sicherstellt. Für diesen Mechanismus sind die Informationen, die die Modifikation des Sprungbefehls erlauben, auszunutzen. Entsprechende Optimierungen für den WCC befinden sich noch in der Entwicklung.

Wie Abb. 5.10 auf der nächsten Seite zeigt, wird ein Updatemechanismus zum Erzeugen einer Loopbound in Abschnitt 5.6.1 vorgestellt. Ferner können Loopbounds – wie in Abschnitt 5.6.2 gezeigt wird – versetzt, d. h. an ein anderes Element (also auch an eine andere Schleife) angebunden werden. Ein letzter Mechanismus, der nur in ICD-C verfügbar ist und in

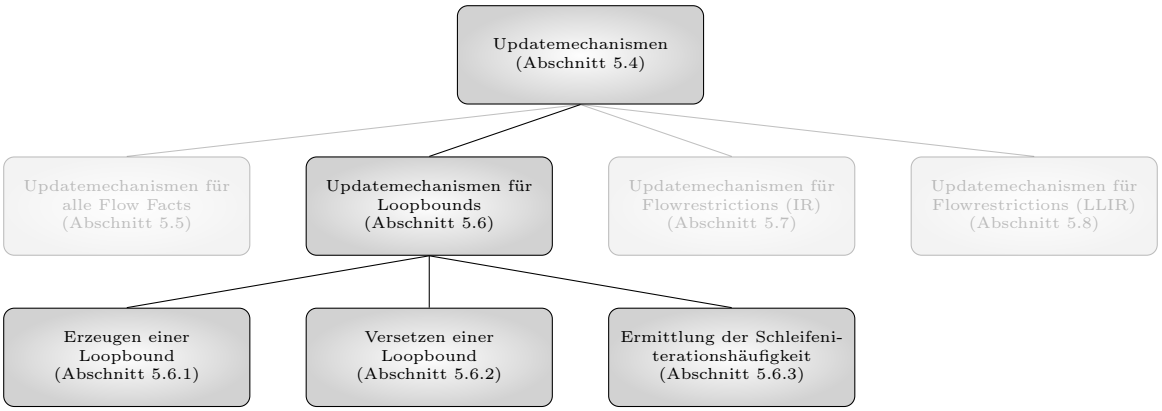


Abbildung 5.10: Updatemechanismen für Loopbounds

Abschnitt 5.6.3 eingeführt wird, unterstützt den Programmierer einer Optimierung bei der Ermittlung der Grenzen der Iterationshäufigkeit einer Schleife.

In Abschnitt 5.6.4 werden abschließend Einsatzszenarien für diese Mechanismen in den Optimierungen präsentiert.

5.6.1 Erzeugen einer Loopbound

Der Updatemechanismus *Erzeugen einer Loopbound* erlaubt für ein Loop Statement der IR bzw. für einen Basisblock der LLIR, eine neue Loopbound zu spezifizieren. Dabei muss der Programmierer einer Optimierung sicherstellen, dass keine Schleife durch zwei Loopbounds annotiert wird, und dass ein Basisblock der LLIR tatsächlich zu einer durch aiT erkennbaren Schleife gehört.

Während die Überprüfung einer einmaligen Annotierung einer Schleife per Loopbound in ICD-C trivial ist, da für jedes annotierbare Loop Statement ohne Loopbound eine neue Loopbound erzeugt werden darf, kann eine Loopbound in der ICD-LLIR theoretisch für jeden Basisblock einer Schleife annotiert werden. Eine Prüfung, welche Basisblöcke genau zu einer Schleife gehören und dementsprechend, ob eine Schleife schon annotiert wurde, ist nicht trivial. Allerdings ist dies – so lange eine Loopbound sicher dem Basisblock mit seiner Abbruchbedingung zugeordnet wird – nicht notwendig, da nur dieser Basisblock mit der Abbruchbedingung zu ermitteln ist.

Der Einsatz dieses Updatemechanismus ist zudem primär auf hoher Abstraktionsebene – also in der IR – zu erwarten, da vor allem auf dieser Abstraktionsebene Schleifen analysiert und optimiert und Loopbounds dadurch annotiert werden.

Zum Erzeugen einer Loopbound muss neben dem Loop Statement bzw. einem entsprechenden Basisblock einer Schleife eine minimale (min) und eine maximale (max) Iterationshäufigkeit bekannt sein. Die Loopbound kann dann durch:

- IR: `flowfactsCreateLoopbound( IR_LoopStmt loop, int min, int max );`

- LLIR: flowfactsCreateLoopbound(LLIR_BB bb, int min, int max);

erzeugt werden.

Fazit

Der Updatemechanismus *Erzeugen einer Loopbound* bietet die Möglichkeit, für eine Schleife – z. B. nach einer Analyse – eine neue Loopbound zu annotieren. In Kombination mit dem Löschen einer alten Loopbound (Abschnitt 5.5.1) kann aber auch die minimale und maximale Iterationshäufigkeit modifiziert werden.

Die Sicherheit und Präzision dieser Technik ist von den Grenzen der neuen Loopbound abhängig. Somit liegt beides in der Verantwortung des Programmierers der Optimierung oder Analyse, die diese Loopbound erzeugt.

5.6.2 Versetzen einer Loopbound

Der Updatemechanismus *Versetzen einer Loopbound* ermöglicht, eine Loopbound von einem Loop Statement der IR zu einem zweiten Loop Statement bzw. von einem Basisblock der LLIR zu einem zweiten Basisblock zu verschieben. Dabei muss in der LLIR dieser Basisblock natürlich wieder zu einer durch aiT ermittelbaren Schleife – aber nicht zwingend zu einer anderen Schleife – gehören. Bei der Verschiebung einer Loopbound bleiben die durch sie spezifizierten Iterationshäufigkeiten unverändert.

Wie schon bei der *Erzeugung einer Loopbound* liegt auch hier die Verantwortung, die mehrfache Annotation einer Schleife durch Loopbounds zu verhindern, beim Programmierer einer Optimierung. Dieser kann durch die Methoden:

- IR: flowfactsMoveLoopbounds(IR_LoopStmt source, IR_LoopStmt target);
- LLIR: flowfactsMoveLoopbounds(LLIR_BB source, LLIR_BB target);

Loopbounds verschieben.

Fazit

Die Sicherheit dieses Updates wird ausführlich in Abschnitt 5.2.3 diskutiert.

Wird z. B. durch eine Optimierung die Bindung zwischen einem Basisblock der LLIR mit einer Loopbound und der Abbruchbedingung der zugehörigen Schleife gelöst, so ließe sich dieser Mechanismus nutzen, um die Loopbound entsprechend zu versetzen und diese Bindung dadurch aufrecht zu erhalten.

5.6.3 Ermittlung der Schleifeniterationshäufigkeit

Die *Ermittlung der Schleifeniterationshäufigkeit* bietet – ohne selbst ein Update von Flow Facts durchzuführen – in ICD-C die Möglichkeit, aus den Analyseergebnissen des zu ICD-C gehörenden Loop Analyzers und aus annotierten Loopbounds eine möglichst präzise minimale und maximale Iterationshäufigkeit für eine Schleife automatisch zu ermitteln.

Wenn ein Loop Analyzer für seine Schleife eine genaue Iterationshäufigkeit ermitteln konnte, so sind diese präzisen Werte zu nutzen. Andernfalls ist nach einer vorhandenen Loopbound zu suchen, um deren (potentiell ungenaueren) Werte weiter zu verwenden.

Gefundene Werte können dabei zur Unterstützung von Optimierungen wie *Loop Unrolling* durch einen Unroll-Faktor geteilt werden. Zur Unterstützung aller anderen Optimierungen ist dieser Wert mit 1 vordefiniert, um die ursprünglich ermittelten Werte zurückzugeben. Wird ein Unroll-Faktor spezifiziert, so wird gemäß der Analyse der Sicherheit von Transformationen für Loopbounds in Abschnitt 5.2.3 auf Seite 53 – wenn die Teilung von *min* und *max* nicht ganzzahlig aufgeht – *min* stets ab- und *max* stets aufgerundet.

Die Auswertung erfolgt durch die Methode:

- IR: `flowfactsDetermineNewNumbersOfIterations( IR_LoopStmt loop, IR_LoopAnalyzer analyzer, int unroll );`

und liefert ein Zahlenpaar `pair<int, int>` für die minimale und maximale Iterationshäufigkeit bzw. für beide Werte -1, wenn keine Werte ermittelt werden konnten.

Fazit

Der Einsatz dieses Mechanismus ist nur in Verbindung mit weiteren Mechanismen sinnvoll, die dessen Ergebnisse nutzen (z. B. das *Erzeugen einer Loopbound*), da er selbst kein Update von Flow Facts bietet, dieses aber unterstützen kann. Der Mechanismus bietet auch keine eigene Analyse einer Schleife, sondern nutzt die Vorgaben durch Loop Analyzer und alte Loopbound. Entsprechend wird durch deren Vorgaben auch die Sicherheit und die Präzision dieses Mechanismus bestimmt.

Bei Bedarf kann diese Methode erweitert werden, um weitere Analyseergebnisse für die Ermittlung der Iterationshäufigkeit einer Schleife zu berücksichtigen.

5.6.4 Einsatzbeispiele

Im Folgenden werden Beispiele für den Einsatz der Updatemechanismen *Erzeugen einer Loopbound*, *Versetzen einer Loopbound* und *Ermittlung der Schleifeniterationshäufigkeit* in Optimierungen vorgestellt.

Erzeugen einer Loopbound

Wird durch eine Optimierung eine neue Schleife angelegt, so kann der Updatemechanismus *Erzeugen einer Loopbound* genutzt werden, um diese Schleife mit einer Loopbound zu annotieren.

Beispiel: *Durch die ICD-C Optimierung Loop Collapsing werden zwei verschachtelte for-Schleifen zu einer neuen gemeinsamen Schleife zusammengefasst, um so den Overhead durch Kontrollflusssprünge zu reduzieren. Da diese Optimierung nur ausgeführt wird, wenn u. a. die genaue Iterationshäufigkeit beider Schleifen ermittelt werden konnte, ergibt sich die Iterationshäufigkeit der neuen Schleife als deren Produkt.*

Ein Aufruf von flowfactsCreateLoopbound(combinedLoop, prod, prod); *mit prod als diesem Produkt annotiert eine sichere und präzise Iterationshäufigkeit für diese neue Schleife.*

Versetzen einer Loopbound

Das *Versetzen einer Loopbound* kann in ICD-C genutzt werden, wenn eine Schleife durch eine zweite Schleife mit gleicher Iterationshäufigkeit ersetzt wird.

Beispiel: *Durch die ICD-C Optimierung Transform Head Controlled Loops wird eine Schleife mit Prüfung der Schleifenabbruchbedingung zu Beginn jeder Iteration durch eine Schleife mit Prüfung dieser Abbruchbedingung am Ende jeder Iteration ersetzt. Loopbounds werden dabei von der alten zur neuen Schleife verschoben, da sich die Iterationshäufigkeit durch die Transformation nicht verändert. Lediglich der Overhead, der durch die Kontrollflusssprünge und Abbruchbedingungsprüfungen, die durch die Schleife verursacht werden, entsteht, wird reduziert.*

Ermittlung der Schleifeniterationshäufigkeit, Erzeugen einer Loopbound

Vor allem in Kombination ermöglichen die Updatemechanismen beliebige Manipulationen von Loopbounds, da ggf. eine alte Loopbound entfernt und durch das anschließende *Erzeugen einer Loopbound* ersetzt werden kann. Dies kann unterstützt durch die *Ermittlung der Schleifeniterationshäufigkeit* oder direkt unter Nutzung von Analyseergebnissen erfolgen.

Beispiel: *Durch die ICD-C Optimierung Loop Unrolling wird die Iterationshäufigkeit einer Schleife um einen Faktor $u > 1$ reduziert. Dementsprechend sind auch die annotierten Iterationshäufigkeiten anzupassen. Da u. U. genaue Iterationshäufigkeiten durch eine Analyse des Loop Analyzers verfügbar sind, wird der Mechanismus zur Ermittlung von Schleifeniterationshäufigkeiten genutzt.*

Anschließend wird die bisherige Loopbound gelöscht und durch eine neue Loopbound mit den soeben ermittelten Werten ersetzt.

5.7 Updatemechanismen für Flowrestrictions (IR)

Durch Flowrestrictions kann in der IR die Ausführungshäufigkeit nahezu aller Statements in beliebiger Kombination zueinander ins Verhältnis gesetzt werden. Diese hohe Flexibilität spiegelt sich in der hohen Komplexität derer Updatemechanismen wieder. Dabei sind die Updates von Flowrestrictions stark von der verursachenden Optimierung abhängig, wie folgende zwei Beispiele zeigen.

Beispiel: *Durch die ICD-C Optimierung Fold Constant Code kann ein if-Statement mit konstant wahrer Bedingung durch seinen then-Anweisungsblock ersetzt werden.*

In diesem Fall ist lediglich jedes Vorkommen der Entscheidungsvariablen für das if-Statement in Flowrestrictions zu ersetzen, jene Vorkommen von Entscheidungsvariablen z. B. für Statements im then-Anweisungsblock können unverändert bleiben.

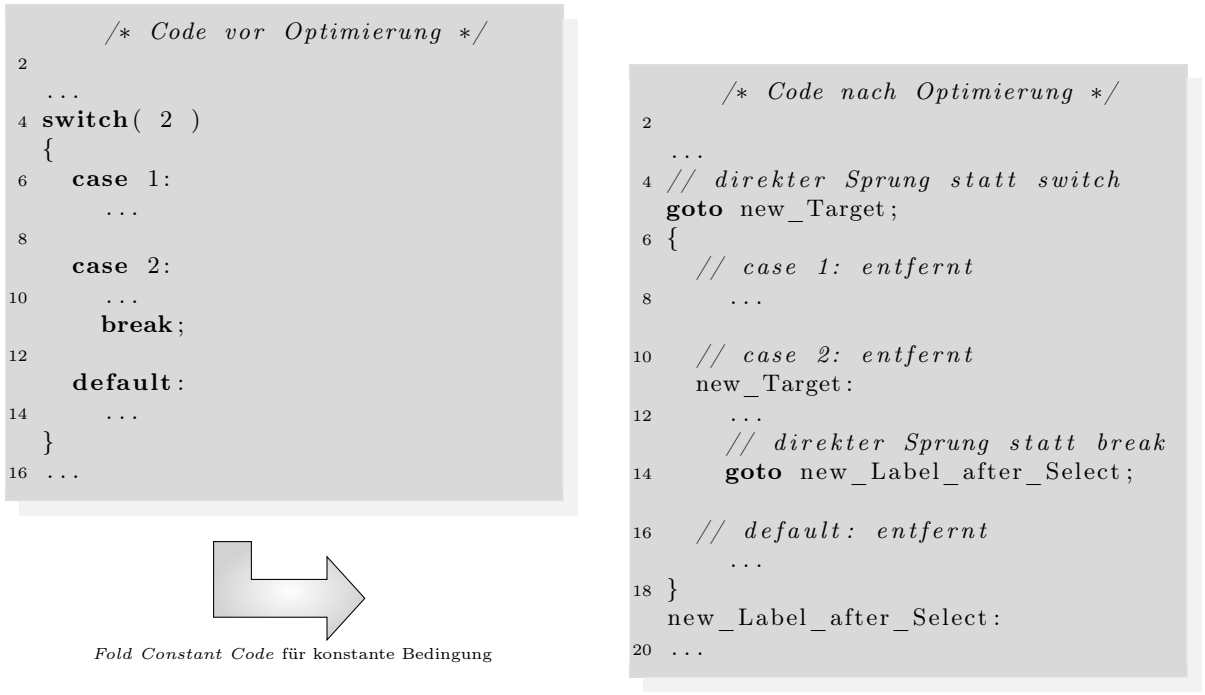


Abbildung 5.11: Bsp. *Fold Constant Code* bei einem *switch*-Statement

Beispiel: *Ein switch-Statement mit konstanter Bedingung wie in Abb. 5.11 kann durch die ICD-C Optimierung Fold Constant Code entfernt werden. Dazu wird der select-Anweisungsblock vom switch-Statement in dessen Eltern Statement verschoben. Unmittelbar nach dem case- bzw. default-Statement, das durch das switch-Statement stets erreicht wurde, wird ein Label Statement eingefügt. Anschließend werden alle case- und default-Statements des switch-Statements entfernt, all seine break-Statements werden durch Sprünge auf ein neues Label Statement unmittelbar nach dem select-Anweisungsblock ersetzt. Zuletzt wird das switch-Statement durch einen Sprung auf das dazu angelegte Label Statement im select-Anweisungsblock ersetzt.*

Für diese Optimierung ist also nicht nur jedes Vorkommen der Entscheidungsvariablen für das switch-Statement in Flowrestrictions zu ersetzen, sondern auch das jener Entscheidungsvariablen für die case-, default- und break-Statements.

Grundlage aller Aktualisierungen von Flowrestrictions ist die Ersetzung von Entscheidungsvariablen durch eine gewichtete Summe von Entscheidungsvariablen. Anstatt für jede Optimierung ein individuelles Update zu entwickeln, wird ein flexibler Ersetzungsmechanismus

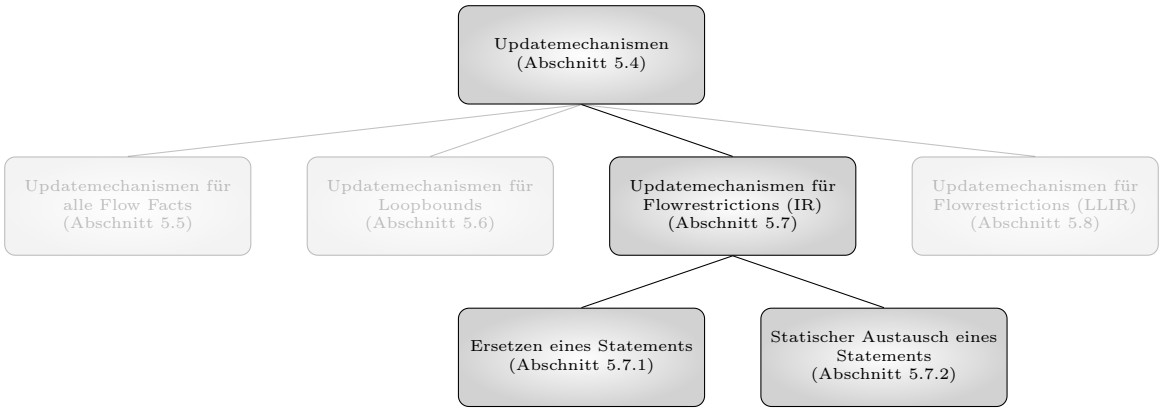


Abbildung 5.12: Updatemechanismen für Flowrestrictions (IR)

in Abschnitt 5.7.1 eingeführt. Wie Abb. 5.12 zeigt, wird dieser Mechanismus lediglich durch einen statischen Austausch – einem Sonderfall der flexiblen Ersetzung für statische Methoden – in Abschnitt 5.7.2 ergänzt.

Ein solcher flexibler Ersetzungsmechanismus erlaubt auch zukünftig entwickelten Optimierungen, ohne Neuentwicklung von Updatemechanismen Flowrestrictions zu aktualisieren. Lediglich eine Analyse der Auswirkung einer neuen Optimierung auf Flowrestrictions wird notwendig sein, um den vorhandenen Mechanismus effizient einzusetzen.

Obwohl Flowrestrictions auf Entscheidungsvariablen für Ausführungshäufigkeiten von Statements (bzw. in der LLIR und in aiT von Basisblöcken) basieren, werden syntaktisch zunächst Referenzen auf deren zugehörige Statements durch sie verwaltet. Deshalb wird in den algorithmengeprägten Betrachtungen dieses Abschnitts i. d. R. vereinfachend von Statements in Flowrestrictions statt von Vorkommen von Entscheidungsvariablen für Statements in Flowrestrictions gesprochen werden.

5.7.1 Ersetzen eines Statements

Der Updatemechanismus *Ersetzen eines Statements* ermöglicht, ein Statement sowohl gleichwertig als auch abschätzend in allen Flowrestrictions zu ersetzen. Er erlaubt die Vorgabe dieser Ersetzungen durch den Programmierer einer Optimierung, bietet unterstützende Mechanismen zu deren Ermittlung sowie eine automatisierte Suche nach Ersetzungen. Er gewährleistet in jedem Fall – notfalls durch das Löschen von Flowrestrictions – eine sichere $\mathrm{WCET}_{\mathrm{est}}$-Berechnung.

Das Verfahren besteht aus vier Schritten für jedes zu ersetzende Statement, wobei der zweite und dritte Schritt optional sind, der zweite aber auch mehrere Male ausgeführt werden kann:

1. **Planen einer Ersetzung**

   Dieser Schritt umfasst insbesondere die Prüfung, ob überhaupt Flowrestrictions durch das entsprechende Statement betroffen sind.

2. **Sammeln möglicher Ersetzungen**

   Dieser Schritt erlaubt dem Programmierer einer Optimierung, mögliche Ersetzungen für ein Statement zu spezifizieren.

3. **Automatisierte Suche möglicher Ersetzungen**

   Dieser Schritt ermittelt automatisch mögliche Ersetzungen für ein Statement.

4. **Durchführen einer Ersetzung**

   Dieser Schritt führt letztendlich die Ersetzungen des Statements in den Flowrestrictions durch (gleichwertige Ersetzungen werden abschätzenden Ersetzungen wegen ihrer Qualität bezüglich der $WCET_{est}$-Berechnung vorgezogen).

Jeder der Schritte wird in den Optimierungen durch entsprechende Methoden angestoßen. Dabei wird eine Ersetzung stets durch das zu ersetzende Statement identifiziert, so dass für mehrere Statements parallel Ersetzungen vorbereitet werden können.

Datencontainer für Updates: Updatedata

Die Daten für das *Ersetzen eines Statements* werden in einem eigenen Datencontainer - *Updatedata* (IR_Updatedata) genannt - gesammelt und umfassen folgende Informationen:

id : Das Statement, für das dieses Updatedata angelegt wurde.

ref : Das zugehörige Flowfactref.

geq : Eine Liste von möglichen abschätzenden Ersetzungen mit höherer Ausführungshäufigkeit (Abschätzungen nach oben).

leq : Eine Liste von möglichen abschätzenden Ersetzungen mit geringerer Ausführungshäufigkeit (Abschätzungen nach unten).

equ : Eine Möglichkeit für eine gleichwertige Ersetzung (genau eine gleichwertige Ersetzung genügt für ein sicheres und präzises Update).

auto: Ein Flag, das anzeigt, ob die automatisierte Suche möglicher Ersetzungen in Phase 3 bereits erfolgte bzw. unterbunden wurde.

Updatedata werden zentral für eine Optimierung per

```
Menge<Paar<IR_Stmt, IR_Updatedata> > ffupdates;
```

verwaltet und durch das Statement, für welches die Daten gesammelt werden, identifiziert.

Phase 1: Planen einer Ersetzung

In der ersten Phase des *Ersetzen eines Statements*, die durch

```
flowfactsPrepareUpdate( IR_Stmt id );
```

angestoßen wird, ist zunächst festzustellen, ob dieses Statement (id) in einer Flowrestriction referenziert wird, und daher ein Update notwendig ist. Nur wenn ein Update notwendig ist, wird für dieses Statement ein neues Updatedata angelegt (vgl. Abb. 5.13 auf der nächsten Seite). In den folgenden Phasen wird daher stets aus der (Nicht-)Existenz eines Updatedatas

```
  flowfactsPrepareUpdate( IR_Stmt id )
2 begin

4   /* Lege Datenstruktur an, wenn Update notwendig. */
    if( hat_Flowrestrictions( id ) ) then
6     ffupdates += neues_Paar( id, neues_Updatedata( id ) );

8 end
```

Abbildung 5.13: Algorithmus: Phase 1: Planen einer Ersetzung

gefolgert, ob Flowrestrictions für ein Statement zu aktualisieren und die Mechanismen daher auszuführen sind.

Phase 2: Sammeln möglicher Ersetzungen

In Phase zwei können gleichwertige und abschätzende Ersetzungen für ein Statement, die in einer Optimierung bekannt sind, gesammelt werden. Zur Klassifizierung einer Ersetzung wird ein Modus (flowfactsmode) benutzt:

equ: Ersetzung ist gleichwertig.

leq : Ersetzung ist abschätzend mit geringerer Ausführungshäufigkeit.

geq: Ersetzung ist abschätzend mit größerer Ausführungshäufigkeit.

Entsprechend des Modus wird eine mögliche Ersetzung eines Statements dem Updatedata hinzugefügt. Die Ersetzungen (Summe gewichteter Statements) können durch Aufruf von

flowfactsAdd(IR_Stmt id, list<pair<int, IR_Stmt> > sum, flowfactsmode m);

flowfactsAdd(IR_Stmt id, int factor, IR_Stmt stmt, flowfactsmode m);

flowfactsAdd(IR_Stmt id, IR_Stmt stmt, flowfactsmode m);

spezifiziert werden. Dabei sind die beiden letzten Varianten abkürzend für genau ein ersetzendes Statement bzw. für genau ein ersetzendes Statement mit Faktor eins.

Ist nur bekannt, dass z. B. ein Compound Statement gleich oft wie ein zu ersetzendes Statement durch den impliziten Kontrollfluss betreten wird, so muss in diesem nach einer passenden Ersetzung gesucht werden. Diesbezüglich sind diverse Szenarien denkbar.

Beispiel: *Das erste Statement im Compound Statement ist ein Expression Statement. Dieses ist – da genau einmal pro Erreichen durch den impliziten Kontrollfluss ausgeführt – eine gleichwertige Ersetzung.*

Beispiel: *Das erste Statement im Compound Statement ist ein Label Statement. Dieses kann – da es neben dem impliziten Kontrollfluss auch jenen durch explizite Sprünge erhält – eine abschätzende Ersetzung nach oben sein.*

Beispiel: *Das erste Statement im Compound Statement ist ein while-Statement. Dieses gehört zu einer neuen Schleife und kann damit weder eine gleichwertige (nach Definition 5.2 auf Seite 52) noch eine abschätzende (nach Definition 5.3 auf Seite 52) Ersetzung sein.*

Entsprechende Analysen müssen nicht in der Optimierung vorgenommen werden. Ein Mechanismus – in den Abschnitten 5.7.1.1 und 5.7.1.2 erläutert – unterstützt den Programmierer einer Optimierung und erlaubt, per

```
flowfactsAddSingle( IR_Stmt id, IR_Stmt stmt, flowfactsmode m );
```

eine Suche z. B. in einem solchen Compound Statement zu starten.

Analog kann ein Basisblock für die Suche einer Ersetzung per

```
flowfactsAddSingle( IR_Stmt id, IR_BasicBlock bb, flowfactsmode m );
```

spezifiziert werden. Der entsprechende Mechanismus wird in Abschnitt 5.7.1.3 dargestellt.

Die Angabe eines Modus erlaubt in beiden Mechanismen, nicht nur durch equ nach einer gleichwertigen Ersetzung zu suchen, sondern durch leq und geq auch nach abschätzenden Ersetzungen zum zu ersetzenden Statement zu suchen. Dies ist möglich, da durch die Mechanismen selbst eine gleichwertige Ersetzung zum spezifizierten Statement stmt bzw. Basisblock bb (und nicht zum zu ersetzenden Statement id) gesucht wird.

Nutzt der Programmierer einer Optimierung z. B. ein Basisblock mit geringerer Ausführungshäufigkeit als das zu ersetzende Statement für den Aufruf des flowfactsAddSingle()-Mechanismus auf Basisblockebene, so muss er nur durch Wahl des Modus leq den richtigen Datencontainer im entsprechenden Updatedata spezifizieren. So kann das Ergebnis der Suche bei einer Ersetzung in Phase vier korrekt genutzt werden kann.

Werden in Phase zwei mehrere gleichwertige Ersetzungen spezifiziert, so wird nur die erste berücksichtigt. Abschätzende Ersetzungen werden dagegen alle im entsprechenden Container des Updatedatas gesammelt.

Phase 3: Automatisierte Suche möglicher Ersetzungen

Phase drei erlaubt – falls noch keine gleichwertige Ersetzung spezifiziert wurde – automatisch nach einer solchen zu suchen oder zumindest (weitere) abschätzende Ersetzungen zu ergänzen. Sie wird höchstens einmal pro *Ersetzen eines Statements* ausgeführt, und kann durch die Optimierung per

```
flowfactsSearchUpdate( IR_Stmt id );
```

angestoßen werden. Alternativ wird sie automatisch zu Beginn der vierten Phase durchgeführt. Dies kann der Programmierer einer Optimierung allerdings durch den Aufruf von

```
flowfactsPreventSearchUpdate( IR_Stmt id );
```

für die Ersetzung eines Statement verhindern.

Der Algorithmus in Abb. 5.14 auf der nächsten Seite nutzt für diese automatisierte Suche weitestgehend dieselben Mechanismen, die für den Einsatz in Optimierungen in Phase zwei zur Verfügung stehen. Die flowfactsAddSingle(IR_Stmt, IR_Stmt, flowfactsmode)-Methode in Zeile 13 (Abb. 5.14) durchsucht vom zu ersetzenden Statement aus dessen Vorgänger und Nachfolger nach gleichwertigen und direkt anschließend nach abschätzenden Ersetzungen. Die flowfactsAddSingle(IR_Stmt, IR_BasicBlock, flowfactsmode)-Methode in Zeile 20 (Abb. 5.14,

```
  flowfactsSearchUpdate( IR_Stmt id )
2 begin

4   /* Prüfe Voraussetzungen. */
    Updatedata update = hole_Updatedata( id );
6     if( !update || update.auto || hat_gleichwertige_Ersetzung( update ) ) then
        return();
8
    /* Vermerke, dass Suche durchgeführt wurde. */
10  update.auto = true;

12  /* Suche Ersetzungen auf Statementebene (Abschnitt 5.7.1.1 und 5.7.1.2). */
    flowfactsAddSingle( id, id, equ );
14
    // Wenn jetzt gleichwertige Ersetzung vorhanden, breche ab.
16  if( hat_gleichwertige_Ersetzung( update ) ) then
      return();
18
    /* Suche gleichwertige Ersetzung auf Basisblockebene (Abschnitt 5.7.1.3). */
20  flowfactsAddSingle( id );

22  // Wenn jetzt gleichwertige Ersetzung vorhanden, breche ab.
    if( hat_gleichwertige_Ersetzung( update ) ) then
24    return();

26  /* Suche abschätzende Ersetzung auf Basisblockebene (Abschnitt 5.7.1.4). */
    getEstimationByBB( update );
28
  end
```

Abbildung 5.14: Algorithmus: Phase 3: Automatisierte Suche möglicher Ersetzungen

der zweite und dritte Parameter sind in der Funktionsdeklaration passend vordefiniert) durchsucht den Basisblock des zu ersetzenden Statements nach gleichwertigen Ersetzungen.

Ergänzt wird eine Suche durch Analyse des Kontrollflusses der Funktion (eine Suche auf Basis von Basisblöcken) nach abschätzenden Ersetzungen (Abb. 5.14, Zeile 27), wie sie in Abschnitt 5.7.1.4 eingeführt wird.

Die Mechanismen werden in Reihenfolge der Qualität ihrer Ersetzungen (gleichwertige vor abschätzenden) und ihres Aufwandes (Statement basierte vor Basisblock basierten) durchgeführt, bis eine gleichwertige Ersetzung gefunden wurde oder alle Möglichkeiten erschöpft sind.

Phase 4: Durchführen einer Ersetzung

Durch den Aufruf von

```
flowfactsDoUpdate( IR_Stmt id );
```

wird die vierte Phase des *Ersetzen eines Statements* initiiert (vgl. Abb. 5.15 auf der nächsten Seite).

```
flowfactsDoUpdate( IR_Stmt id )
begin

  /* Wenn Update geplant, aktualisiere globale Updatedatasammlung. */
  Updatedata update = hole_Updatedata( id );
  if( !update ) then
    return();
  ffupdates -= update;

  /* Initiire automatisierte Suche. */
  flowfactsSearchUpdate( id );

  /* 1.) Versuche gleichwertige Ersetzung. */
  if( hat_gleichwertige_Ersetzung( update ) ) then
  begin
    for fr in hole_Flowrestrictions( id ) do
      führe_Ersetzung_durch( fr, id, update.equ ); // siehe Abb. 5.16
    return();
  end

  /* 2. Erhalte transitive Informationen (Abschnitt 5.7.1.5). */
  flowfactsAddTransitiveInformations( id );

  /* 3.) Führe abschätzende Ersetzungen nach oben bzw. unten durch. */
  if( hat_nach_oben_abschätzende_Ersetzung( update ) ) then
  begin
    for fr in hole_Flowrestrictions( id ) do
      if( ist_Summand_der_geq_Seite( fr, id ) ) then
      begin
        for abschätzung in update.geq do
          führe_Ersetzung_durch( erzeuge_Kopie( fr ), id, abschätzung );
        entferne_Flow_Fact( fr );
      end else
      begin
        for abschätzung in update.leq do
          führe_Ersetzung_durch( erzeuge_Kopie( fr ), id, abschätzung );
        entferne_Summand( fr, id ); // abschätzende Ersetzung durch Null
      end
    return();
  end

  /* 4.) Erhalte Sicherheit. */
  for fr in hole_Flowrestrictions( id ) do
    if( ist_Summand_der_geq_Seite( fr, id ) ) then
      entferne_Flow_Fact( fr );
    else
    begin
      for abschätzung in update.leq do
        führe_Ersetzung_durch( erzeuge_Kopie( fr ), id, abschätzung );
      entferne_Summand( fr, id ); // abschätzende Ersetzung durch Null
    end

end
```

Abbildung 5.15: Algorithmus: Phase 4: Durchführen einer Ersetzung

Bevor jedoch die gesammelten Daten für das *Ersetzen eines Statements* genutzt werden, wird durch den Aufruf von flowfactsSearchUpdate() (Abb. 5.15, Zeile 11) die automatisierte Suche möglicher Ersetzungen angestoßen. Diese wird allerdings nur ausgeführt, wenn sie nicht bereits zuvor manuell angestoßen oder in der Optimierung für dieses zu ersetzende Statement untersagt wurde, und wenn zudem noch keine gleichwertige Ersetzung gefunden wurde (die Überprüfungen erfolgen in der entsprechenden Methode). Anschließend wird die Ersetzung durchgeführt.

Ist eine gleichwertige Ersetzung möglich, wird diese ausgeführt und das Update erfolgreich beendet (Abb. 5.15, ab Zeile 13).

Andernfalls werden vor abschätzenden Ersetzungen Kontrollflussinformationen, die durch die Transitivität von Flowrestrictions implizit gegeben sind, explizit zu den Flow Facts des Programms hinzugefügt (Abb. 5.15, Zeile 22). Der entsprechende Mechanismus wird in Abschnitt 5.7.1.5 vorgestellt.

Sind abschätzende Ersetzungen nach oben möglich (nach unten kann immer durch Null abgeschätzt werden), so werden gemäß den Erkenntnissen aus Abschnitt 5.2.1 alle Vorkommen des Statements in Flowrestrictions auf der „größer gleich Seite“ nach oben und all jene Vorkommen auf der „kleiner gleich Seite“ nach unten abgeschätzt (Abb. 5.15, ab Zeile 24). Alternativ können nur die Abschätzungen nach unten durchgeführt werden, so dass Flowrestrictions, die eine Abschätzung nach oben benötigen, zu löschen sind, um die Sicherheit der $WCET_{est}$-Berechnung zu gewährleisten (Abb. 5.15, ab Zeile 42).

Die Durchführung der eigentlichen Ersetzung in einer Flowrestriction wird in Abb. 5.16 skizziert.

```
führe_Ersetzung_durch( IR_Flowrestriction fr , IR_Stmt id ,
                       Liste<Paar<int , IR_Stmt> > abschätzung )
begin

  bool war_geq_Seite = ist_Summand_der_geq_Seite( fr , id );
  int alter_faktor = entferne_Summand( fr , id );

  for Summand in abschätzung do
    if( war_geq_Seite ) then
      addiere_zu_geq( fr , alter_faktor * Summand.first , Summand.second );
    else
      addiere_zu_leq( fr , alter_faktor * Summand.first , Summand.second );

end
```

Abbildung 5.16: Algorithmus: Phase 4: Ersetzung in einer Flowrestriction

Direktes Ersetzen

Abkürzend lassen sich die vier Schritte – wenn eine gleichwertige Ersetzung sicher bekannt ist – durch den Aufruf von

flowfactsDoUpdate(IR_Stmt id, int factor, IR_Stmt target);

durchführen. Dabei wird in allen Flowrestrictions das Statements id gleichwertig durch das Statement target ersetzt (Abb. 5.17).

```
  flowfactsDoUpdate( IR_Stmt id, int factor, IR_Stmt target )
2 begin

4   flowfactsPrepareUpdate( id );
    flowfactsAdd( id, factor, target, equ );
6   flowfactsDoUpdate( id );

8 end
```

Abbildung 5.17: Algorithmus: direktes Ersetzen eines Statements

Parallele Vorbereitung

Häufig gliedern sich Optimierungen in mehreren Phasen: z. B. wird in einer ersten Phase geprüft, ob eine Optimierung ausgeführt werden kann. In einer zweiten Phase werden wichtige Parameter (wie z. B. der Unroll-Faktor für ein *Loop Unrolling*) ermittelt, aber auch von der Optimierung peripher tangierte Elemente der Datenstruktur angepasst. In einer letzten Phase wird dann die Optimierung selbst durchgeführt.

In diesen Phasen werden u. U. an verschiedenen Stellen Statements, für die ein Update notwendig ist (oft in der ersten Phase), aber auch mögliche Ersetzungen (oft in der zweiten oder letzten Phase) gefunden. Um nun nicht für jedes tangierte Statement nacheinander eine Ersetzung vorbereiten, (manuell oder automatisiert) nach möglichen Ersetzungen suchen und die Ersetzung schließlich durchführen zu müssen, können Updates für mehrere Statements parallel vorbereitet werden. Das *Ersetzen eines Statements* wird so optimal in Optimierungen integriert, ohne deren etablierte Abläufe modifizieren zu müssen.

Abhängigkeiten von Ersetzungen

Gegenseitige Abhängigkeiten in der Datenstruktur können Seiteneffekte für die Ersetzungen von mehreren Statements haben und sind durch den Programmierer einer Optimierung zu bedenken.

Beispiel: *Wird je ein Update für Statement* A *und Statement* B *(sequentiell oder parallel) vorbereitet, und sind die beiden Statements gegenseitig gleichwertige Ersetzungen, so kann z. B. erst* A *durch* B *und dann* B *durch* A *ersetzt werden. Es verbleiben letztendlich Flowrestrictions mit Bezug auf* A*, was i. d. R. nicht der Intention des Programmierers einer Optimierung entsprechen dürfte.*

Diese Abhängigkeiten untereinander können durch präzise Vorgaben von Ersetzungen in Phase zwei aufgelöst werden, während vor allem automatisierte Suchen potentiell zu diesem nicht erwünschten Verhalten führen können.

Unterstützende Mechanismen

In den folgenden Abschnitten werden die bereits erwähnten Mechanismen im Detail vorgestellt. Das sind

- die Suche einer gleichwertigen Ersetzung auf Statementebene in Abschnitt 5.7.1.1,
- die Suche einer abschätzenden Ersetzung auf Statementebene in Abschnitt 5.7.1.2,
- die Suche einer gleichwertigen Ersetzung auf Basisblockebene in Abschnitt 5.7.1.3,
- die Suche einer abschätzenden Ersetzung auf Basisblockebene in Abschnitt 5.7.1.4,
- sowie die Erweiterung von Flow Facts durch transitive Informationen in Abschnitt 5.7.1.5.

5.7.1.1 Suche einer gleichwertigen Ersetzung auf Statementebene

In ICD-C wird der Kontrollfluss eines Programms durch die semantischen Informationen der Statements präziser beschrieben als durch Basisblöcke, so dass die Gleichwertigkeit von Statements nicht auf ein intuitives „liegen im gleichen Basisblock und werden daher auf dieselbe Entscheidungsvariable für aiT abgebildet“ beschränkt werden muss. Statt dessen sind die zwei Faktoren

1. werden gleich oft ausgeführt und
2. gehören zur gleichen Schleife

für die Gleichwertigkeit von Statements ausschlaggebend (vgl. Abschnitt 5.2.2).

```
...
while( Bed_1 ){
  ExpStmt A;
  while( Bed_2 ){
    ExpStmt B;
  }
  ExpStmt C;
}
...
```

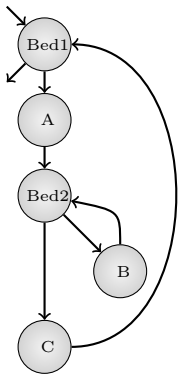


Abbildung 5.18: Bsp. gleichwertiger Statements

Beispiel: *Wird in Abb. 5.18 durch eine Optimierung Statement* A *entfernt, so sind Flowrestrictions mit Bezug auf dieses Statement zu aktualisieren. Offensichtlich werden Statement* A *und Statement* C *gleich oft ausgeführt und gehören zudem zur selben Schleife. Obwohl beide Statements nicht im selben Basisblock liegen, ist Statement* C *eine gleichwertige Ersetzung für Statement* A*.*

```
  flowfactsAddSingle( IR_Stmt id, IR_Stmt stmt, flowfactsmode m )
2 begin

4   /* Prüfe Vorbedingungen. */
    Updatedata update = hole_Updatedata( id );
6   if( !update || hat_gleichwertige_Ersetzung( update ) ) then
      return(); // nichts zu tun
8
    /* Expression Stmts des for-Stmts können nicht behandelt werden. */
10  if( !ist_CompoundStmt( stmt ) &&
        ist_ForStmt( hole_Eltern_Stmt( stmt ) ) ) then
12    return();

14  /* Ermittle Daten für Suche. */
    if( ist_CompoundStmt( stmt ) ) then
16    succ = hole_erstes_Statement( stmt );
    else
18    succ = hole_Nachfolger( stmt );
    pred = hole_Vorgänger( stmt );
20
    /* Führe Suchen durch und speichere Ergebnis. */
22  IR_Stmt ergebnis = 0;
    ergebnis = suche_Vorwärts( succ, hole_Eltern_Stmt( succ ) );
24  if( !ergebnis ) then
      ergebnis = suche_Rückwärts( pred, hole_Eltern_Stmt( pred ) );
26  if( !ergebnis ) then
      return(); // kein Erfolg
28
    if( m == equ ) then
30    update.equ = ergebnis;
    else if ( m == leq ) then
32    update.leq += ergebnis;
    else if ( m == geq ) then
34    update.geq += ergebnis;

36 end
```

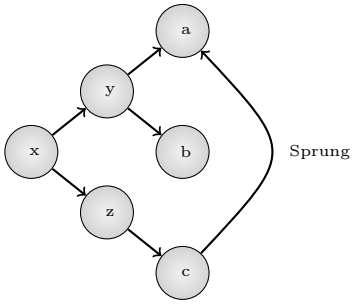
Abbildung 5.19: Algorithmus: Suche einer gleichwertigen Ersetzung auf Statementebene

In diesem Abschnitt wird ein Verfahren (Abb. 5.19) vorgestellt, das in zwei Suchen für ein Statement (stmt) – einer Vorwärtssuche für dessen Nachfolger und einer Rückwärtssuche für dessen Vorgänger – die Semantik von Statements zur Beurteilung nutzt, ob ein gefundenes Statement gleichwertig zu diesem ist, oder ob zumindest über dieses Statement hinaus weiter nach einer gleichwertigen Ersetzung gesucht werden kann.

Dafür sind für ein Statement dessen unmittelbarer Nachfolger und Vorgänger im impliziten Kontrollfluss zu ermitteln (Abb. 5.19, ab Zeile 14), um mit diesen eine Vorwärts bzw. Rückwärtssuche zu starten (Abb. 5.19, ab Zeile 21). Die Ergebnisse der Suchen werden abschließend für ein Ersetzung gespeichert (Abb. 5.19, ab Zeile 29).

Zur Beurteilung der Semantik eines Statements ist insbesondere wichtig, welche effektive Auswirkung es auf den Kontrollfluss hat. Vor allem bei explizit modellierten Sprüngen – also jenen

in Verbindung mit einem Jump oder Targeted Statement – können die effektiven Auswirkungen sehr unterschiedlich sein. Folgende Klassifizierung von Sprüngen und Statements wird bei der Entwicklung eines Algorithmus helfen.



- Sprung bzgl. x lokal → x lokales Statement
- Sprung bzgl. y nicht lokal → y kein lokales Statement
- Sprung bzgl. z nicht lokal → z kein lokales Statement
- ...

Abbildung 5.20: Bsp. lokaler Sprünge/Statements

Definition 5.4
Ein Sprung in der Abarbeitung von Statements ist bezüglich eines Statements x **lokal**, wenn sowohl das den Sprung verursachende Statement als auch dessen Ziel Nachfahren von x in der Statement Hierarchie oder x selbst sind. Ein Statement x heiße **lokal**, wenn kein Nachfahre von x an einem nicht lokalen Sprung bzgl. x partizipiert (Beispiel vgl. Abb. 5.20).

Die Nachfahren eines lokalen Statements ändern den effektiven Kontrollfluss nicht, so dass die direkten Vorgänger und Nachfolger eines lokalen Statements potentiell gleich oft ausgeführt werden. Ist ein Statement aber nicht lokal, so ist eine Suche nach gleichwertigen Ersetzungen unmittelbar zu beenden, da für die in der Suche folgenden Statements nicht mehr die gleiche Ausführungshäufigkeit garantiert werden kann.

Semantik von Statements

Für den Entwurf der Vorwärts- und Rückwärtssuche ist nun die Semantik der verschiedenen ICD-C Statements zu untersuchen. Vergleiche der Ausführungshäufigkeiten zum Vorgänger sind auf die Vorwärtssuche und jene zum Nachfolger auf die Rückwärtssuche – unter der Voraussetzung, dass jener Vorgänger bzw. Nachfolger lokal ist – zu verstehen.

- Compound Statement

  Ein Compound Statement darf nicht in Flow Facts referenziert werden. Der Kontrollfluss wird an sein erstes Kind weitergegeben (ein Compound Statement verwaltet beliebig viele Statements, die entsprechend ihrer Reihenfolge im Compound Statement – sofern nicht durch die Semantik eines anderen Statements beeinflusst – sequentiell ausgeführt werden). Deswegen sind die Kinder gemäß der Suchrichtung von vorne bzw. von hinten an nach einer gleichwertigen Ersetzung zu durchsuchen.

- Expression Statement

  Ein Expression Statement wird genauso oft wie seine impliziten Vorgänger und Nachfolger im Kontrollfluss ausgeführt. Es ist daher bei beiden Suchen eine gleichwertige Ersetzung. Die Suche wird folglich erfolgreich beendet.

- Function Load Argument Statement

  Dieses Statement dient nur dem Laden von Funktionsargumenten und wird von der entsprechenden Funktion verwaltet. Es kann nicht durch eine Suche erreicht werden.

- Jump Statement (*break*-, *continue*-, *goto*- und *return*-Statements)

  Durch ein Jump Statement wird der Kontrollfluss an ein anderes als das – durch dessen Vorfahren vorgegebene – implizit nachfolgende Statement übergeben. Daher wird das Jump Statement zwar genauso oft wie sein impliziter Vorgänger ausgeführt, steht aber nicht direkt in Relation zur Ausführungshäufigkeit seines impliziten Nachfolgers. Bei einer Vorwärtssuche ist es eine gleichwertige Ersetzung. Eine Rückwärtssuche ist dagegen erfolglos abzubrechen.

- Targeted Statement (*case*-, *default*- und Label Statements)

  Ein Targeted Statement erhält neben dem Kontrollfluss seines impliziten Vorgängers auch möglicherweise den von *goto*- oder *switch*-Statements, wird also häufiger als sein Vorgänger, aber genauso oft wie sein Nachfolger ausgeführt. Folglich ist ein Targeted Statement für eine Rückwärtssuche eine gleichwertige Ersetzung, eine Vorwärtssuche muss dagegen erfolglos abgebrochen werden.

- Selection Statement (*if-else*-, *if*- und *switch*-Statements)

  Ein Selection Statement bietet eine temporäre Verzweigung im Kontrollfluss. Es wird genauso oft wie sein Vorgänger ausgeführt und ist so eine gleichwertige Ersetzung bei einer Vorwärtssuche. Ist das Selection Statement lokal (da es Nachfahren hat, ist eine entsprechende Prüfung notwendig), so ist es auch bei der Rückwärtssuche eine gleichwertige Ersetzung. Andernfalls ist die Suche erfolglos abzubrechen. Die Nachfahren eignen sich wegen ihrer bedingten Ausführung i. d. R. nicht als gleichwertige Ersetzung.

- Loop Statement (*for*-, *while*- und *do-while*-Statements)

  Da ein Loop Statement zu einer neuen Schleife gehört, kann es keine gleichwertige Ersetzung sein. Ist das Loop Statement lokal, so kann die Suche über dieses hinaus fortgesetzt werden, sonst ist die Suche erfolglos abzubrechen. Die Nachkommen eines Loop Statements gehören ebenfalls zu einer neuen Schleife und können daher keine gleichwertige Ersetzung sein mit Ausnahme des Init Clause Statements einer *for*-Schleife, das in seiner Eigenschaft als Compound Statement zu handhaben ist.

  Für den Vergleich der Ausführungshäufigkeit einer Schleife zu der weiterer Vorgänger und Nachfolger gilt, dass die Schleife einmal ausgeführt wird, während ihr Inhalt inkl. der Abbruchbedingung durchaus diverse Iterationen durchlaufen kann.

Wird in der Vorwärtssuche ein Expression, Jump oder Selection Statement gefunden, ist die Suche erfolgreich beendet. Bei einem Compound Statement ist in dessen Nachfahren von vorne und bei einem lokalen Loop Statement über dies hinaus weiterzusuchen. Bei einem Targeted Statement oder einem nicht lokalen Loop Statement ist die Suche erfolglos abzubrechen.

Ähnlich ist das Bild bei der Rückwärtssuche: Wird ein Expression, Targeted oder ein lokales Selection Statement gefunden, so ist die Suche erfolgreich beendet. Bei einem Compound Statement ist in dessen Nachfahren von hinten und bei einem lokalen Loop Statement über dies hinaus weiterzusuchen. Bei einem Jump Statement oder einem nicht lokalen Selection oder nicht lokalen Loop Statement ist die Suche erfolglos abzubrechen.

Tabelle 5.1 fasst die Ergebnisse dieser Analyse von Statements in den Spalten Vorwärtssuche und Rückwärtssuche zusammen.

ICD-C Element	Vorwärtssuche	Rückwärtssuche	Vorfahren
Compound Statement	Nachfahren	Nachfahren	weiters.
Expression Statement	✓	✓	-
Function Load Argument Statement	-	-	-
Jump Statement	✓	Abbruch	-
Targeted Statement	Abbruch	✓	-
Selection Statement	✓	lokal? ✓: Abbruch	Abbruch
Loop Statement	lokal? weiters. : Abbruch	lokal? weiters. : Abbruch	Abbruch
Function	-	-	Abbruch

Legende:

✓	Suche erfolgreich beendet.
Abbruch	Suche erfolglos beendet.
weiters.	Fahre Suche mit Nachfolger bzw. Vorgänger fort.
Nachfahren	Durchsuche Nachfahren gemäß der Suchrichtung.
Vorfahren	Durchsuche Vorfahren gemäß der Suchrichtung.
lokal ? A : B	Wenn Statement lokal, dann A, sonst B. (analog zur bedingten Anweisung)
-	nicht möglich

Tabelle 5.1: Übersicht Statementsemantik für Suche gleichwertiger Ersetzungen

Ist durch die Vorwärts- bzw. Rückwärtssuche ein Compound Statement bis zum Ende bzw. Anfang durchsucht worden, stellt sich die Frage, ob in dessen Vorfahren weitergesucht werden kann. Die vorangegangene Analyse legt nahe, dies nur zu erlauben, wenn der direkte Vorfahre auch ein Compound Statements ist (Tabelle 5.1 Spalte Vorfahren).

Algorithmen

Die Analyseergebnisse sind nun zur Entwicklung von Algorithmen zur Vorwärts- und Rückwärtssuche (Abb. 5.21 auf der nächsten Seite, Rückwärtssuche analog) sowie zur Lokalitätsprüfung (Abb. 5.22 auf Seite 84) zu nutzen.

Die Vorwärtssuche wird mit einem Start-Statement und dessen Compound Statement aufgerufen. Der Fall eines Statements ohne Compound Statement – dies kann nur das Init Clause Statement, die Abbruchbedingung oder der Folgeausdruck eines *for*-Statements sowie das Top Compound Statement einer Funktion sein – wird bereits zuvor abgefangen (Abb. 5.19 auf Seite 79, ab Zeile 9). Individuelle Lösungen dieser Sonderfälle ließen sich dort entwickeln.

In dem Compound Statement werden nun beginnend mit dem Start-Statement gemäß der vorangegangenen Analyse die Statements bzgl. ihrer Eignung als gleichwertige Ersetzung überprüft (Abb. 5.21, ab Zeile 4). Wurde das Compound Statement ohne Erfolg und Abbruch durchlaufen, wird – wenn dessen Eltern Statement ebenfalls ein Compound Statement ist – in diesem fortgefahren (Abb. 5.21, ab Zeile 35).

Zuvor wird lediglich ein Sonderfall geprüft (Abb. 5.21, ab Zeile 31). Ist das Compound Statement der Schleifenkörper einer *for*-Schleife, so kann dessen Folgeausdruck (ein Expression Statement) gleichwertige Ersetzung sein.

Zur Prüfung der Lokalität eines Statements ist für jedes Jump oder Targeted Statement in seinen Nachfahren zu prüfen, ob dessen Sprungpartner – also Ziel bzw. Ursprung des Sprungs

```
IR_Stmt suche_Vorwärts( IR_Stmt stmt, IR_CompoundStmt comp )
begin

  /* Suche Vorwärts. */
  for s in comp, beginnend mit stmt do
  begin
    if( ist_CompoundStmt( s ) ) then
    begin
      comp = s;
      s = hole_erstes_Statement( comp );
    end

    if( ist_ExpressionStmt( s ) || ist_JumpStmt( s ) ||
        ist_SelectionStmt( s ) ) then
      return( s );

    if( ist_TargetedStmt( s ) ) then
      return( 0 );

    if( ist_LoopStmt( s ) ) then
    begin
      if( ist_ForStmt( s ) && !ist_leer( hole_InitClause( s ) ) ) then
        return( hole_ExprStmt( hole_InitClause( s ) ) );
      if( !isLocalStatement( s ) ) then // siehe Abb. 5.22
        return( 0 );
    end
  end

  IR_Stmt eltern_stmt = hole_Eltern_Stmt( comp );

  /* Folgeausdruck der for-Schleife gehört zum Schleifenkörper. */
  if( ist_ForStmt( eltern_stmt ) && hat_ContStmt( eltern_stmt ) ) then
    return ( hole_ContStmt( eltern_stmt ) );

  /* Weitersuche über Eltern Statement. */
  if( ist_CompoundStmt( eltern_stmt ) ) then
    return ( suche_Vorwärts( hole_Nachfolger ( comp ), eltern_stmt ) );

  return( 0 );

end
```

Abbildung 5.21: Algorithmus: Vorwärtssuche

– auch wieder zu den Nachfahren des vermeintlich lokalen Statements gehört, oder dies gar selbst ist. Falls dies für einen Nachfahren nicht zutrifft, ist das Statement nicht lokal.

Dabei gilt: *Return*-Statements sind nie Nachfahre eines lokalen Statements. Da die Suche nach Vorgängern bzw. Nachfolgern von *goto*- und Label Statements rechenaufwändig ist (diese Vorgänger und Nachfolger werden nicht explizit in der ICD-C verwaltet, sondern sind durch eine Kontrollflussanalyse oder eine manuelle Suche zu ermitteln), wird auch aus ihnen unmittelbar die Nicht-Lokalität eines Statements geschlussfolgert.

```
bool isLocalStatement( IR_Stmt stmt )
begin

  Menge<IR_Stmt> stmts = Nachfahren( s ) ∪ stmt;

  for s in Nachfahren( stmt ) do
    if( ist_BreakStmt( s ) &&
          !hole_abgebrochenes_Loop_oder_Selection_Stmt( s ) in stmts ||
        ist_ContinueStmt( s ) && !hole_Schleife( s ) in stmts ||
        ist_GotoStmt( s ) ||
        ist_ReturnStmt( s ) ||
        ( ist_CaseStmt( s ) || ist_DefaultStmt( s ) ) &&
          !hole_Switch( s ) in stmts ||
        ist_Label_Stmt( s ) ) then
      return( false );

  return( true );

end
```

Abbildung 5.22: Algorithmus: Lokalitätsprüfung

Fazit

Bei geringem Rechenaufwand – da vor allem auf eine Kontrollflussanalyse verzichtet wird – bietet diese Suche auf Statementebene die Möglichkeit, über Kreise und Verzweigungen im Kontrollfluss hinweg gleichwertige Ersetzungen zu finden, da z. B. lokale Loop oder Selection Statements während der Suche übersprungen werden.

Es sind aber noch diverse Erweiterungen der vorgestellten Mechanismen denkbar:

- Durch die Nutzung einer Summe könnte z. B. bei der Rückwärtssuche eine gleichwertige Ersetzung in den beiden Anweisungsblöcken eines *if-else*-Statements gefunden werden, da sicher einer der beiden Blöcke durchlaufen werden muss, um das implizit folgende Statement zu erreichen.
- Zur Lokalitätsprüfung könnte eine Kontrollflussanalyse genutzt werden, um die Sprünge zwischen *goto*- und Label Statements genauer zu analysieren, statt vom Worst-Case Szenario eines nicht lokalen Sprungs auszugehen. Dies würde allerdings den Rechenzeit-Vorteil gegenüber den Basisblock basierenden Suchen egalisieren.
- ...

Letztendlich bietet die Suche auf Statementebene aber schon jetzt hohe Erfolgsaussichten bei der Suche einer gleichwertigen Ersetzung.

5.7.1.2 Suche einer abschätzenden Ersetzung auf Statementebene

Die Algorithmen aus Abschnitt 5.7.1.1 zur Suche gleichwertiger Ersetzungen auf Statementebene können erweitert werden, so dass in bisher erfolglosen Fällen i. d. R. zumindest abschät-

zende Ersetzungen gefunden werden.

- Erweiterung der Vorwärts/Rückwärtssuche

  Die Ausführungshäufigkeit eines Targeted Statement bei der Vorwärtssuche ist höchstens größer als die des ursprünglichen Statements, gehört zur selben Schleife und ist somit eine abschätzende Ersetzung nach oben. Ebenso ist auch ein Jump Statement bei einer Rückwärtssuche eine abschätzende Ersetzung nach oben.

- Erweiterung der Lokalitätsprüfung

  Auch wenn ein nicht lokaler Sprung bei der Lokalitätsprüfung eines Statements gefunden wird, so muss die Suche nicht abgebrochen werden, da Targeted Statements bei der Vorwärtssuche die Ausführungshäufigkeit höchstens erhöhen, Jump Statements dagegen allenfalls reduzieren (bei der Rückwärtssuche invers).

  Solange nur Erhöhungen oder Reduktionen der Ausführungshäufigkeit beobachtet werden, ist das nächste Statement, das unter Vernachlässigung der Lokalitätsbedingung eine gleichwertige Ersetzung wäre, eine abschätzende Ersetzung nach oben bzw. nach unten. Nur wenn sowohl Erhöhung als auch Reduktion der Ausführungshäufigkeit beobachtet werden, ist die Suche erfolglos abzubrechen, da ihr Effekt in Summe nicht bekannt ist.

Anpassen der Algorithmen

Da nicht nur gleichwertige sondern auch abschätzende Ersetzungen durchgeführt werden können, ist zu vermerken, ob die nächste gefundene Ersetzung gleichwertig (`equ`), abschätzend mit geringerer Ausführungshäufigkeit (`leq`) oder abschätzend mit höherer Ausführungshäufigkeit (`geq`) ist. Dieser Wert ist mit dem Modus zu initialisieren, mit dem die Suche auf Statementebene in Abb. 5.19 auf Seite 79 aufgerufen wird und in allen betroffenen Algorithmen verfügbar zu machen.

Bei der Erweiterung der Vorwärtssuche in Abb. 5.21 auf Seite 83 ist im Falle eines Targeted Statements – sofern der Modus `equ` oder `geq` ist – dieses zurückzugeben und der Modus ggf. auf `geq` zu ändern. Andernfalls ist die Suche nach wie vor erfolglos abzubrechen. Die Rückwärtssuche ist analog für Jump Statements anzupassen.

Für die Vorwärts- und die Rückwärtssuche werden nun eigene Lokalitätsprüfungen benötigt. Für die Vorwärtssuche ist diese Prüfung aus Abb. 5.22 auf der vorherigen Seite so zu verändern, dass sie bei Targeted Statements nur noch die Suche abbricht, wenn der Modus `leq` war. Ansonsten ist der Modus auf `geq` zu ändern, da darauf folgende Statements höchstens häufiger ausgeführt werden. Ebenso ist bei Jump Statements der Modus – sofern er nicht `geq` war und die Suche daher abzubrechen ist – auf `leq` zu ändern, da darauf folgende Statements höchstens seltener ausgeführt werden. Wieder sind die Auswirkungen von Jump und Targeted Statements für die Lokalitätsprüfung der Rückwärtssuche invers.

Letztendlich ist das Ergebnis dieser erweiterten Vorwärts als auch Rückwärtssuche jeweils dem Container im Updatedata zuzuordnen, der durch den gemeinsamen Modus spezifiziert wird.

Fazit

Bei geringem Rechenaufwand werden Szenarien, die zu einem Abbruch der Suche nach einer gleichwertigen Ersetzung auf Statementebene führen, für eine abschätzende Ersetzung in Betracht gezogen, so dass diese – falls insgesamt keine gleichwertige Ersetzung möglich ist – für eine abschätzende Ersetzung zur Verfügung stehen.

5.7.1.3 Suche einer gleichwertigen Ersetzung auf Basisblockebene

Statements, die in einem Basisblock liegen, sind bzgl. der WCET_{est}-Berechnung trivialerweise gleichwertig. Diese Tatsache wird in der *Suche einer gleichwertigen Ersetzung auf Basisblockebene* genutzt.

```
  flowfactsAddSingle( IR_Stmt id, IR_BasicBlock bb = 0, flowfactsmode m = equ )
2 begin

4   /* Prüfe Vorbedingungen. */
    Updatedata update = hole_Updatedata( id );
6   if( !update || hat_gleichwertige_Ersetzung( update ) ) then
      return(); // nichts zu tun
8
    /* Falls kein Basisblock definiert ist, nutze den vom Statement. */
10  if ( !bb ) then
      bb  = hole_BB( id );
12
    /* Suche im Basisblock nach einem Statement ungleich id. */
14  for stmt in bb do
      if( stmt != id ) then
16    begin
        /* Speichere Ergebnis. */
18      if( m == equ ) then
          update.equ = stmt;
20      else if ( m == leq ) then
          update.leq += stmt;
22      else if ( m == geq ) then
          update.geq += stmt;
24      return();
      end
26
  end
```

Abbildung 5.23: Algorithmus: Gleichwertige Ersetzung auf Basisblockebene

Der Algorithmus – in Abb. 5.23 skizziert – ist simpel, da lediglich der entsprechende Basisblock nach einem von id verschiedenen Statement durchsucht werden muss. Dabei wird – falls die Methode ohne Basisblock aufgerufen wird – der Basisblock des entsprechenden Statements durchsucht. Das Ergebnis ist dann dementsprechend gleichwertig.

Fazit

Die Stärke des Mechanismus liegt darin, dass er die etablierte Kontrollflussanalyse von ICD-C nutzt und erfolgreich ist, sobald ein Basisblock mehr als ein Statement enthält (bei Nutzung in Phase zwei des *Ersetzens eines Statements* durch eine Optimierung u. U. gar bei nur einem Statement), ist allerdings wegen der notwendigen Kontrollflussanalyse auch entsprechend rechenaufwändig.

5.7.1.4 Suche einer abschätzenden Ersetzung auf Basisblockebene

In diesem Abschnitt wird ein Verfahren vorgestellt, das die Kontrollflussanalyse zur Ermittlung abschätzender Ersetzungen nutzt.

Nach ICD-Cs Kontrollflussanalyse für ein Programm sind u. a. folgende Informationen verfügbar:

- X ist die Menge der Basisblöcke,
- $pred : X \to \wp(X)$ ist die Abbildung von Basisblöcken auf deren unmittelbare Vorgänger,
- $succ : X \to \wp(X)$ ist die Abbildung von Basisblöcken auf deren unmittelbare Nachfolger.

Zusätzlich sei

- $\# : X \to \mathbb{N}_0$ eine Abbildung eines Basisblocks auf dessen Ausführungshäufigkeit, wie sie letztendlich durch aiT vorgenommen wird.

Für einen Basisblock $\mathsf{bb} \in X$ gilt dann:

$$\#(bb) \leq \sum_{p \in pred(bb)} \#(p) \qquad \text{sowie} \tag{5.15}$$

$$\#(bb) \leq \sum_{s \in succ(bb)} \#(s). \tag{5.16}$$

Da die Vorgänger von bb neben bb auch weitere Nachfolger haben können, muss deren Ausführung nicht zwingend die von bb folgen. Auf der anderen Seite geht aber jeder Ausführung von bb die Ausführung eines seiner Vorgänger voraus. Daher ist die Summe der Ausführungshäufigkeiten der Vorgänger eines Basisblocks stets größer gleich der Ausführungshäufigkeit dieses Basisblocks. Die Menge der Vorgänger ($pred(bb)$) eignet sich dementsprechend als eine abschätzende Ersetzung nach oben. Selbes gilt analog auch für die Menge der Nachfolger ($succ(bb)$) eines Basisblocks.

Wird nun jedoch von der Summe der Ausführungshäufigkeiten der Vorgänger eines Basisblocks die Summe der Ausführungshäufigkeiten derer Nachfolger ohne diesen Basisblock abgezogen, so ist diese Differenz höchstens so groß wie die Ausführungshäufigkeit des Basisblocks, da

der Ausführung der Nachfolger der Vorgänger des Basisblocks nicht zwingend die Ausführung eines seiner Vorgänger vorausgehen muss. Folglich gilt:

$$\#(bb) \geq \sum_{p \in pred(bb)} \#(p) - \sum_{s \in succ(pred(bb)) \setminus \{bb\}} \#(s) \qquad \text{sowie} \qquad (5.17)$$

$$\#(bb) \geq \sum_{s \in succ(bb)} \#(s) - \sum_{p \in pred(succ(bb)) \setminus \{bb\}} \#(p). \qquad (5.18)$$

So sind abschätzende Ersetzungen nach unten gefunden.

Grenzen des Verfahrens

Zwei Faktoren müssen in dieser Betrachtung allerdings noch ergänzt werden:

- Basisblock ist sein eigener Vorgänger/Nachfolger

  Ist ein Basisblock sein eigener Vorgänger (und damit auch automatisch sein eigener Nachfolger), so sind die Aussagen in (5.15) bis (5.18) weiterhin gültig. Da aber dieser Mechanismus in Phase drei einer *Ersetzung eines Statements* nur aufgerufen wird, wenn im Basisblock bb nur das Statement id liegt (ansonsten wäre zuvor schon bei der *Suche einer gleichwertigen Ersetzung auf Basisblockebene* erfolgreich eine Ersetzung gefunden worden), wird so das Statement id durch sich selbst abgeschätzt. Folglich verbleiben auch nach dem *Ersetzen eines Statements* Referenzen von Flowrestrictions auf dieses Statement.

  Da dies dem Ziel des Updatemechanismus widerspricht, ist die *Suche einer abschätzenden Ersetzung auf Basisblockebene* in diesem Fall erfolglos abzubrechen.

- Vorgänger oder Nachfolger in Kontrollfluss sind nicht modelliert

  Die Kontrollflussanalyse für ICD-C wird für je eine Funktion durchgeführt. Der erste Basisblock einer Funktion (Funktionseinstieg) kann daher durch den Aufruf der Funktion zur Ausführung kommen, ohne dass einer seiner Vorgänger zwingend zuvor ausgeführt wurde. Der Kontrollfluss ist durch Funktionsaufrufe hindurch nicht modelliert.

  Die Vorgänger eines Funktionseinstiegs eigenen sich dementsprechend nicht für eine Nutzung nach (5.15) bis (5.18). Gleiches gilt für die Nachfolger eines Funktionsausstiegs.

Algorithmus

Der resultierende Algorithmus – in Abb. 5.24 auf der nächsten Seite skizziert – muss dementsprechend neben der Ermittlung der entsprechenden Basisblockmengen prüfen, ob der Basisblock bb durch sich selbst abgeschätzt wird (Abb. 5.24, ab Zeile 13), und ob wegen eines Funktionseinstiegs oder -ausstiegs einzelne Ersetzungen nicht möglich sind (Abb. 5.24, ab Zeile 17).

Für die Speicherung der Ergebnisse (Abb. 5.24, ab Zeile 35) ist natürlich aus jedem Basisblock ein Statement zu nutzen, wobei die Statements, deren Basisblockausführungshäufigkeit abzuziehen ist, mit dem Faktor -1 in die Summe eingehen.

```
getEstimationByBB( IR_Updatedata update )
begin

  IR_BasicBlock bb = hole_BB( update.id );

  /* Bilde Basisblockmengen. */
  Menge<IR_BasisBlock> pred_bbs, succ_bbs, succpred_bbs, predsucc_bbs;
  pred_bbs = pred( bb );
  succ_bbs = succ( bb );
  succpred_bbs = succ( pred_bbs ) \ { bb };
  predsucc_bbs = pred( succ_bbs ) \ { bb };

  /* Prüfe, ob bb sein eigener Vorgänger/Nachfolger ist. */
  if( bb in pred_bbs ) then
    return();

  /* Prüfe auf Funktionseinstieg und -ausstieg. */
  Menge<IR_BasicBlock> entry, exits;
  entry = hole_BB_des_Funktionseinstiegs( hole_Funktion( bb ) );
  exits = hole_BBs_der_Funktionsausstiege( hole_Funktion( bb ) );

  // Prüfe auf Vollständigkeit der Vorgänger.
  if( bb in entry ) then
    pred_bbs = succpred_bbs = ∅;
  if( succ_bbs ∩ entry != ∅ ) then
    predsucc_bbs = ∅;

  // Prüfe auf Vollständigkeit der Nachfolger. */
  if( bb in exits ) then
    succ_bbs = predsucc_bbs = ∅;
  if( pred_bbs ∩ exits != ∅ ) then
    succpred_bbs = ∅;
  end if

  /* Speichere Ergebnisse, ignoriere leere Mengen. */
  // Aus jedem Basisblock ein Statement, summiert,
  // aus zweiter Basisblockmenge mit Faktor -1.
  update.geq += erzeuge_Abschätzung( pred_bbs );
  update.geq += erzeuge_Abschätzung( succ_bbs );
  update.leq += erzeuge_Abschätzung( pred_bbs, succpred_bbs );
  update.leq += erzeuge_Abschätzung( succ_bbs, predsucc_bbs );

end
```

Abbildung 5.24: Algorithmus: Abschätzende Ersetzung auf Basisblockebene

Fazit

Die Möglichkeit der Methode für nahezu jedes Statement eine abschätzende Ersetzung zu finden, empfiehlt diese für die automatisierte Suche des *Ersetzens eines Statements*. Da eine durch sie ermittelte abschätzende Ersetzung aber beliebig unpräzise sein kann, wird sie nur in letzter Instanz eingesetzt.

5.7.1.5 Erweiterung von Flow Facts durch transitive Informationen

Flowrestrictions sind nach Modell (4.2) lineare Ungleichungen. Als solche sind sie u. a. transitiv, d. h., gilt $A \leq B$ und $B \leq C$, so folgt auch $A \leq C$.

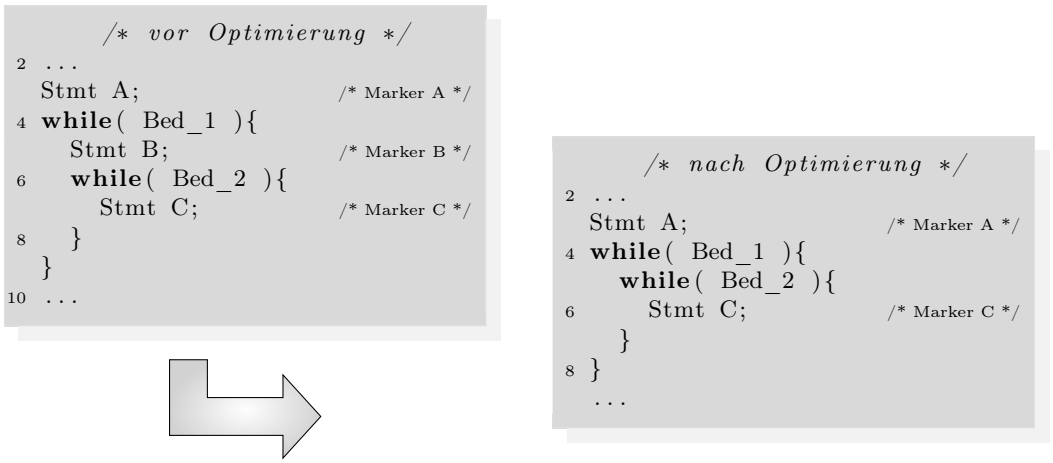


Abbildung 5.25: Anwendungsfall für Erweiterung um transitive Informationen

Beispiel: *Werden für den Codeauschnitt in Abb. 5.25 vor der Optimierung zwei Flowrestrictions* 1*C ≤ 5*B *und* 1*B ≤ 5*A *spezifiziert, so folgt wegen derer Transitivität einer Annotierung von* 1*C ≤ 25*A*, die auch noch nach der Optimierung gilt.*

Durch abschätzende Ersetzungen werden Statements der „kleiner gleich Seite“ nach unten, und jene der „größer gleich Seite“ nach oben abgeschätzt. So gehen diese impliziten transitiven Informationen verloren. Dieser Mechanismus erlaubt vor der abschätzenden Ersetzung, alle durch ein Statement ermöglichten transitiven Schlussfolgerungen zu ermitteln, und diese den Flow Facts des Programms explizit hinzuzufügen.

Algorithmus

Gegeben ist ein Statement id. Die Menge der Flowrestrictions, die id auf der „kleiner gleich Seite“ referenzieren, sei LEQ(id), die mit Referenzen auf der „größer gleich Seite“ entsprechend GEQ(id). Gemäß Modell (4.2) gilt:

$$LEQ(id) \cap GEQ(id) = \emptyset, \tag{5.19}$$

da jedes Statement höchstens einmal pro Flowrestriction referenziert werden kann.

Für alle Paare $p = (x, y)$ aus LEQ(id)×GEQ(id) kann eine neue Flowrestriction durch ineinander Einsetzen ohne id gebildet werden, die nicht durch die abschätzenden Ersetzungen für id beeinflusst wird. Der Algorithmus dazu wird in Abb. 5.26 auf der nächsten Seite skizziert.

Fazit

Gerade wenn ineinander verschachtelte Schleifen wie in Abb. 5.25 durch Flowrestrictions annotiert werden, und keine gleichwertige Ersetzung für das verbindende Element gefunden

```
flowfactsAddTransitiveInformations( IR_Stmt id )
begin

  for p=(x,y) in LEQ(id) × GEQ(id) do
  begin
    /* Erzeuge temporäre Kopien der Flowrestrictions. */
    cx = erzeuge_Kopie( x );
    cy = erzeuge_Kopie( y );

    /* Stelle Ungleichungen nach id um. */
    löse_Flowrestriction_auf_nach( cx, id ); // factor_A*id <= SUM_A
    löse_Flowrestriction_auf_nach( cy, id ); // SUM_B <= factor_B*id

    /* Gleiche Faktoren von id an. */
    int fact_cx = hole_Faktor_von( cx, id ); // factor_A
    int fact_cy = hole_Faktor_von( cy, id ); // factor_B

    mult_Flowrestriction( cy, fact_cx );
    // Ergibt: factor_A*SUM_B <= factor_A*factor_B*id
    mult_Flowrestriction( cx, fact_cy );
    // Ergibt: factor_B*factor_A*id <= factor_B*SUM_A

    /* Erzeuge neue Flowrestriction. */
    erzeuge_neue_Flowrestriction( hole_leq( cy ), hole_geq( cx ) );
    // Ergibt: factor_A*SUM_B <= factor_B*SUM_A

    /* Lösche temporäre kopien. */
    entferne_Flow_Fact( cx );
    entferne_Flow_Fact( cy );
  end

end
```

Abbildung 5.26: Algorithmus: Erweiterung von Flow Facts durch transitive Informationen

wird, vermag die Ausnutzung transitiver Informationen das Qualitätsniveau der WCET_{est}-Berechnung halten zu helfen.

5.7.2 Statischer Austausch eines Statements

Der *statische Austausch eines Statements* erlaubt, alle Referenzen auf ein Statement id durch Referenzen auf ein Statement target auszutauschen. Damit ist dies kein neuer Updatemechanismus, denn das *Ersetzen eines Statements* aus Abschnitt 5.7.1 vermag dasselbe. Es ist ein Spezialfall dessen und entspricht effektiv dem Aufruf von flowfactsDoUpdate(id, 1, target);.

Entstanden ist dieser Mechanismus – der durch den Aufruf von

flowfactsStaticExchange(IR_Stmt id, IR_Stmt target);

ausgeführt wird – für die statischen Methoden der Klasse IR_Transform. Als statische Methoden steht ihnen das *Ersetzen eines Statements* nicht zur Verfügung. Da die Methoden aber

für Schleifen erlauben, *continue*- und *break*-Statements durch *goto*-Statements zu ersetzen, ist ein Update von Flowrestrictions (Loopbounds sind nicht betroffen) notwendig. Die Ersetzungen sind gleichwertig, daher genügt ein statischer Mechanismus, der in den Flowrestrictions Referenzen auf ein Statement durch jene auf ein anderes Statement austauscht.

5.7.3 Einsatzbeispiel

Während der Einsatz des *statischen Austauschs* in Optimierungen trivial ist, soll ein einfaches Beispiel hier verdeutlichen, wie der Mechanismus *Ersetzen eines Statements* zum Update von Flowrestrictions genutzt werden kann.

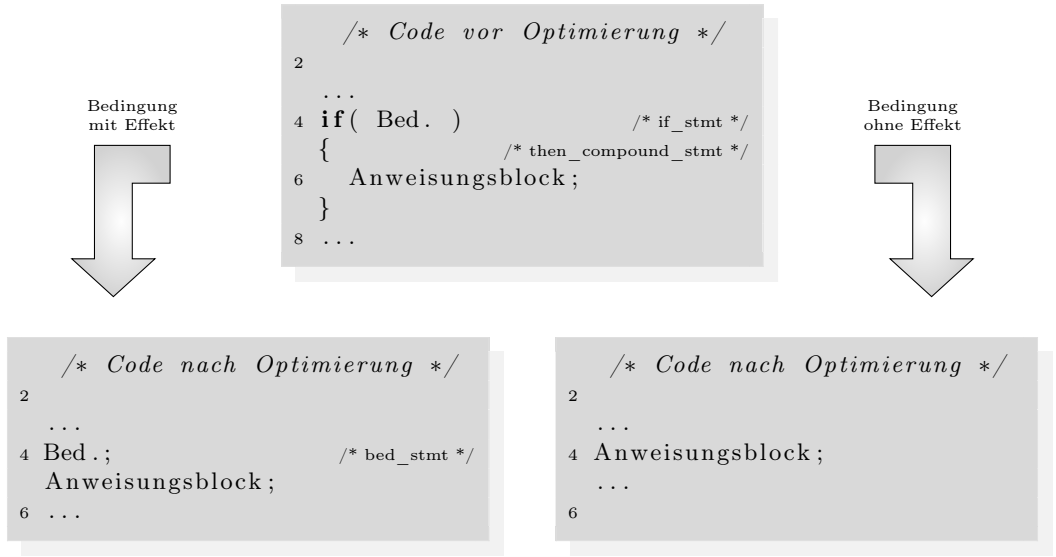


Abbildung 5.27: Bsp. *Fold Constant Code* mit wahrer Bedingung für *if*-Statements

Beispiel: *Gegeben ist ein if-Statement, dessen Bedingung stets wahr ist. Die Modifikation durch die ICD-C Optimierung Fold Constant Code wird in Abb. 5.27 dargestellt. Ein Update der Flowrestrictions kann durch das Ersetzen eines Statements wie folgt erreicht werden:*

1. flowfactsPrepareUpdate(if_stmt);

   Falls Flowrestrictions für das if-Statement existieren, wird ein Updatedata zum Sammeln der Informationen für eine Ersetzung angelegt. Andernfalls werden dieser, aber auch alle folgenden Schritte unmittelbar abgebrochen.

2. flowfactsAdd(if_stmt, bed_stmt, equ);

   Falls die Bedingung des if-Statements einen Effekt hat (indem sie z. B. eine Zuweisung oder ein Inkrement beinhaltet), wird sie unmittelbar vor dem if-Statement als Expression Statement in die Datenstruktur eingefügt. Sie ist eine sichere und präzise Ersetzung und als solche zu vermerken.

3. flowfactsAddSingle(if_stmt, then_compound_stmt, equ);

   Der then-Anweisungsblock des if-Statements, der unmittelbar vor dem if-Statement in die Funktion der IR eingefügt wird, wird wegen der konstant wahren Bedingung genau so oft durch den impliziten Kontrollfluss betreten wie das if-Statement aufgeführt wird. Da bei einer automatisierten Suche nicht die Nachfahren eines Selection Statements durchsucht werden, wird durch diesen Aufruf eine Suche in dessen Nachfahren manuell gestartet. Diese wird allerdings nur ausgeführt, wenn nicht im Schritt vorher schon eine gleichwertige Ersetzung gefunden wurde.

4. flowfactsDoUpdate(if_stmt);

   Falls noch immer keine gleichwertige Ersetzung möglich ist, wird die automatisierte Suche angestoßen. Anschließend werden die Flowrestrictions aktualisiert, so dass das if-Statement letztendlich in keinem Flow Fact (Loopbounds sind nicht möglich) mehr referenziert wird. Es kann nun aus der IR entfernt werden.

5.8 Updatemechanismen für Flowrestrictions (LLIR)

Die grundsätzlichen Überlegungen zur Entwicklung von Updatemechanismen für Flowrestrictions in der LLIR entsprechen jenen aus Abschnitt 5.7 für ICD-C. Jedoch sind zwei wesentliche Unterschiede festzuhalten:

- geänderte Basis für Flowrestrictions

  In ICD-C basieren die Flowrestrictions auf Ausführungshäufigkeiten von Statements, in der ICD-LLIR basieren diese dagegen auf Ausführungshäufigkeiten von Basisblöcken (die Abstraktion von Statements zu Basisblöcken findet in der Translation der Flowrestrictions von ICD-C nach ICD-LLIR statt, vgl. Abschnitt 5.3.2 auf Seite 55). Ohne Statements mit entsprechender Semantik in der LLIR sind dort auch keine entsprechenden Suchen auf Statementebene wie in ICD-C möglich. Da durch den Algorithmus zur Suche einer gleichwertigen Ersetzung auf Basisblockebene nur innerhalb eines Basisblocks nach Ersatz gesucht wird, entfällt auch dieser Mechanismus bei Formulierung der Flowrestrictions auf Basis von Basisblöcken.

- primär Assemblercode Optimierungen

  In der LLIR werden insbesondere die Assemblerbefehle und Befehlsfolgen durch Optimierungen wie die *Peephole Optimization* oder die *Loop Invariant Code Motion* modifiziert. Diese haben i. d. R. keine Auswirkungen auf Flow Facts. So werden z. Z. Flow Facts nur durch die *Empty Basicblock Elimination* beeinflusst.

Soll ICD-Cs *Ersetzen eines Statements* hier analog als ein *Ersetzen eines Basisblocks* entwickelt werden, so verbleiben im Bereich der unterstützenden Mechanismen und automatisierten Suchen lediglich zwei Techniken für eine abschätzende Ersetzung: die Suche abschätzender Ersetzungen auf Basisblockebene sowie die Erweiterung von Flow Facts durch transitive Informationen.

Da in der betroffenen LLIR Optimierung bei jedem Einsatz der Updatemechanismen feststeht, ob eine gleichwertige oder eine abschätzende Ersetzung durchzuführen ist (in der Optimierung

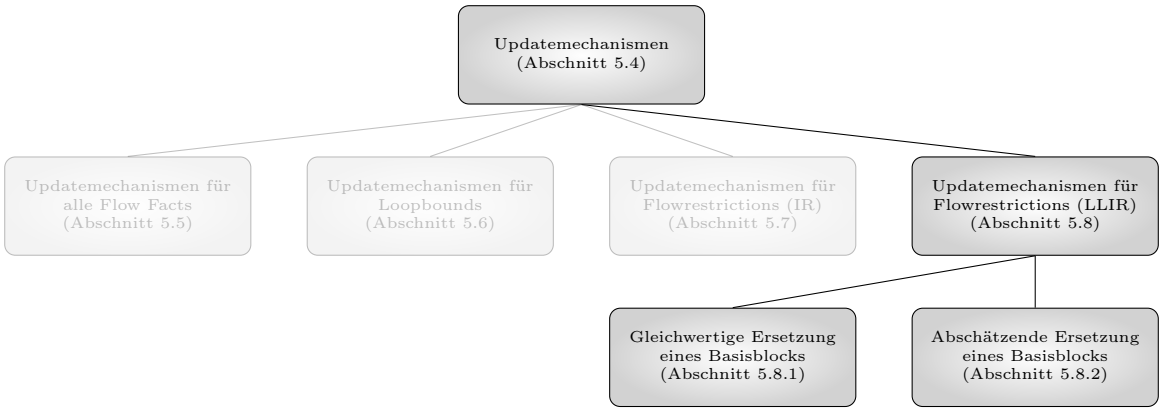


Abbildung 5.28: Updatemechanismen für Flowrestrictions (LLIR)

wird bereits der Kontrollfluss analysiert), können – wie Abb. 5.28 zeigt – zwei getrennt Updatemechanismen entwickelt werden.

Eine gleichwertige Ersetzung von Basisblöcken in Flowrestrictions wird in Abschnitt 5.8.1, eine abschätzende Ersetzung in Abschnitt 5.8.2 vorgestellt. Abschließend wird in Abschnitt 5.8.3 nicht nur der Einsatz dieser Mechanismen demonstriert, sondern auch eine vollständige Analyse einer Optimierung bezüglich ihres Einflusses auf Flowrestrictions vorgestellt.

5.8.1 Gleichwertige Ersetzung eines Basisblocks

Die *gleichwertige Ersetzung eines Basisblocks* in Flowrestrictions erlaubt, Referenzen auf einen Basisblock durch Referenzen auf einen zweiten Basisblock zu ersetzen.

```
  flowfactsExchangeBB( LLIR_BB bb, LLIR_BB bbnew )
2 begin

4   /* Prüfe, ob Flowrestrictions betroffen sind. */
    if( !hat_Flowrestrictions( bb ) ) then
6     return();

8   /* Ersetze Referenzen in Flowrestrictions. */
    for fr in hole_Flowrestrictions( bb ) do
10    tausche_Basisblock_aus( fr, bb, bbnew );

12 end
```

Abbildung 5.29: Algorithmus: gleichwertige Ersetzung eines Basisblocks (LLIR)

Sie wird durch Aufruf von

flowfactsExchangeBB(LLIR_BB bb, LLIR_BB bbnew);

ausgeführt und ist in Abb. 5.29 skizziert.

Fazit

Dieser Mechanismus ist genau dann sicher, wenn die beiden Basisblöcke bb und bbnew gleich oft ausgeführt werden.

5.8.2 Abschätzende Ersetzung eines Basisblocks

Die *abschätzende Ersetzung eines Basisblocks* erlaubt, Referenzen auf einen Basisblock nach oben und unten abschätzend zu ersetzen. Da bei ihrem derzeitigen Einsatz in der ICD-LLIR Optimierung *Empty Basicblock Elimination* stets nur eine abschätzende Ersetzung nach oben aber keine nach unten bekannt ist (vgl. Abschnitt 5.8.3) , wird durch den Aufruf von

flowfactsEstimateBB(LLIR_BB bb, LLIR_BB bbnew);

bisher auch nur die Spezifizierung einer abschätzenden Ersetzung nach oben unterstützt. Eine triviale abschätzende Ersetzung nach unten ist aber durch Null immer gegeben.

```
  flowfactsEstimateBB( LLIR_BB bb, LLIR_BB bbnew )
2 begin

4   /* Prüfe, ob Flowrestrictions betroffen sind. */
    if( !hat_Flowrestrictions( bb ) ) then
6     return();

8   /* Erhalte transitive Informationen. */
    flowfactsAddTransitiveInformations( bb );
10
    /* Ersetze Referenzen in Flowrestrictions. */
12  for fr in hole_Flowrestrictions( bb ) do
      if( ist_Summand_der_geq_Seite( fr, bb ) ) then
14      tausche_Basisblock_aus( fr, bb, bbnew );
      else
16      entferne_Summand( fr, bb );

18 end
```

Abbildung 5.30: Algorithmus: abschätzende Ersetzung eines Basisblocks (LLIR)

Nach der obligatorischen Prüfung, ob ein Update für Flowrestrictions notwendig ist (Abb. 5.30, ab Zeile 4), werden ggf. die transitiven Informationen analog zum ICD-C Mechanismus in Abschnitt 5.7.1.5 explizit zu den Flow Facts des Programms hinzugefügt (Abb. 5.30, ab Zeile 8). Abschließend werden die abschätzenden Ersetzungen nach oben (durch bbnew) und nach unten (durch Null) ausgeführt (Abb. 5.30, ab Zeile 11).

Fazit

Sofern die Ausführungshäufigkeit von bbnew größer gleich der von bb ist, ist dieser Mechanismus sicher.

Erweitert werden kann er, indem – analog zu ICD-C, vgl. Abschnitt 5.7.1.4 – durch eine automatisierte Suche abschätzender Ersetzungen auf Basisblockebene weitere Abschätzungen nach oben und unten ermittelt werden, um u. U. die Qualität der Abschätzung insgesamt zu steigern.

5.8.3 Einsatzbeispiel & Analyse einer Optimierung

Im Folgenden wird der Einsatz der Updatemechanismen *Gleichwertige Ersetzung eines Basisblocks* und *Abgeschätzte Ersetzung eines Basisblocks* in der ICD-LLIR Optimierung *Empty Basicblock Elimination* inklusive der Analyse dieser Optimierung demonstriert.

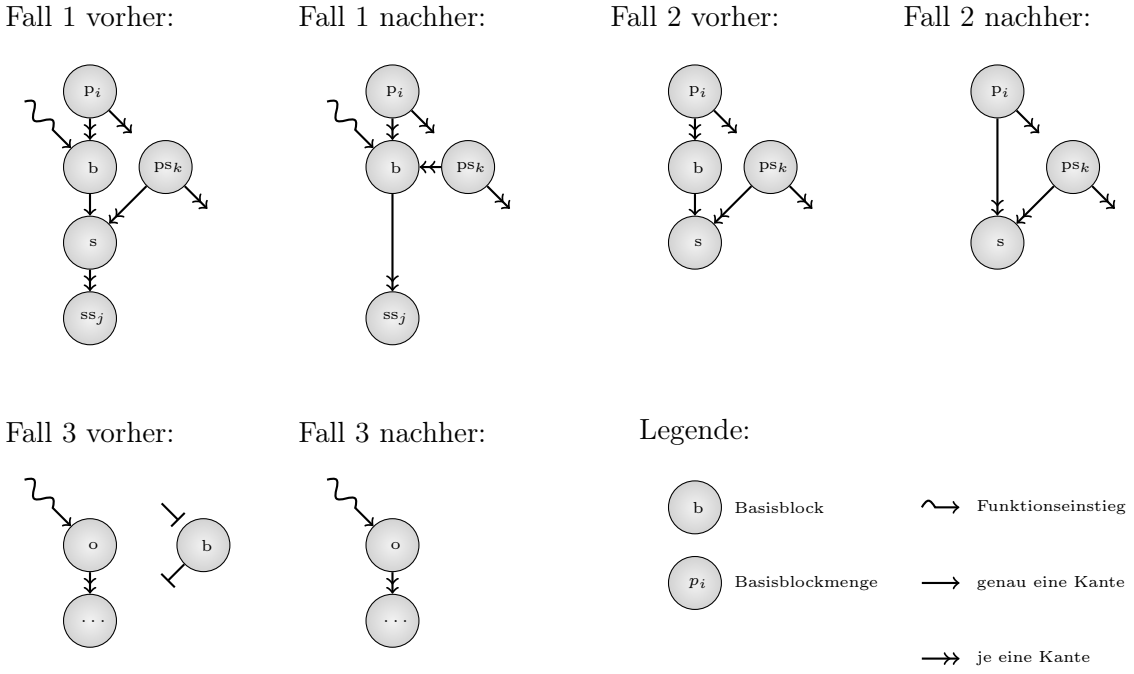


Abbildung 5.31: Analyse *Empty Basicblock Elimination*

Ziel der Optimierung ist – wie der Name schon sagt – Basisblöcke ohne Inhalt zu entfernen. Da ein Basisblock ohne Instruktion insbesondere an seinem Ende keinen Sprung enthält, ist sein Nachfolger eindeutig durch den impliziten Kontrollfluss bestimmt. Der Kontrollfluss kann daher modifiziert werden, um den leeren Basisblock zu löschen. Dazu werden drei Fälle in der Optimierung unterschieden, die entsprechend auch getrennt bzgl. ihrer Auswirkungen auf Flowrestrictions zu analysieren sind:

1. der leere Basisblock hat genau einen Nachfolger und ist der Funktionseinstieg;
2. der leere Basisblock hat genau einen Nachfolger und ist nicht der Funktionseinstieg;
3. der leere Basisblock hat weder Vorgänger noch Nachfolger und ist weder letzter Basisblock seiner Funktion noch Funktionseinstieg.

Die Analyse basiert auf folgendem Modell:

- X ist die Menge der Basisblöcke,
  - b ist leerer Basisblock,
  - s ist der eindeutige Nachfolger von b,
  - p_i mit $i \in I$ ist ein Vorgänger von b,
  - ss_j mit $j \in J$ ist ein Nachfolger des eindeutigen Nachfolgers von b,
  - ps_k mit $k \in K$ ist ein Vorgänger des eindeutigen Nachfolgers von b (ungleich b),
  - o steht zu b in keiner Beziehung;
- $\rightarrow$: $X \cup \{\varepsilon\} \times X$ ist eine Relation, die die Kanten im Kontrollflussgraphen beschreibt (für eine präzise Analyse der Optimierung ist die Modellierung der Kanten des Kontrollflussgraphen notwendig, jedoch können diese – gemäß der Designentscheidungen in Kapitel 4 – nicht zur Formulierung von Flow Facts genutzt werden; $\varepsilon \rightarrow b$ beschreibt einen Funktionseinstieg);
- $\#$: $X, X \cup \{\varepsilon\} \times X \rightarrow \mathbb{N}_0$ ist eine Abbildung von Basisblöcken bzw. von Kanten zwischen Basisblöcken auf deren Ausführungshäufigkeit.

In Abb. 5.31 auf der vorherigen Seite wird der Kontrollflussgraph aller drei Fälle der Optimierung vorgestellt. Diese Fälle sind nun im Einzelnen zu analysieren.

Fall 1: Eindeutiger Nachfolger & Funktionseinstieg

Der leere Basisblock b ist Funktionseinstieg und hat genau einen Nachfolger. Da der Funktionseinstieg nicht entfernt werden darf, der Nachfolger s jedoch eindeutig ist, kann dieser mit b verschmolzen werden, so dass letztendlich s entfällt. Dementsprechend sind Flow Facts, die sich auf die Basisblöcke b und s beziehen, betroffen.

Vor der Optimierung gilt:

$$\#(s) = \#(b \rightarrow s) + \sum_{k \in K} \#(ps_k \rightarrow s) \qquad \text{s. Abb. 5.31} \qquad (5.20)$$

$$\#(b) = \#(\varepsilon \rightarrow b) + \sum_{i \in I} \#(p_i \rightarrow b) \qquad \text{s. Abb. 5.31} \qquad (5.21)$$

$$\#(b) = \#(b \rightarrow s) \qquad \text{da } |succ(b)| = 1 \qquad (5.22)$$

Aus (5.20) - (5.22) folgt:

$$\#(s) = \#(\varepsilon \rightarrow b) + \sum_{i \in I} \#(p_i \rightarrow b) + \sum_{k \in K} \#(ps_k \rightarrow s) \qquad (5.23)$$

Durch die Optimierung ist sichergestellt, dass folgende Beziehung gilt:

$$\sum_{k \in K} \#(ps_k \rightarrow s) = \sum_{k \in K} \#(ps_k \rightarrow b) \qquad (5.24)$$

Nach der Optimierung gilt:

$$\#(b) = \#(\varepsilon \rightarrow b) + \sum_{i \in I} \#(p_i \rightarrow b) + \sum_{k \in K} \#(ps_k \rightarrow b) \qquad \text{s. Abb. 5.31} \qquad (5.25)$$

Vorkommen von Basisblock s können durch Basisblock b ersetzt werden, denn nach (5.24) ist die Ausführungshäufigkeit von s vor der Optimierung (5.23) und von b nach dieser (5.25) gleich. Die Ersetzung von s durch b ist daher gleichwertig.

(5.21) und (5.25) zeigen jedoch, dass die Ausführungshäufigkeit von b um $\sum_{k \in K} \#(ps_k \rightarrow p)$ bzw. $\sum_{k \in K} \#(ps_k \rightarrow b)$ erhöht wurde. Nur wenn $|ps| = 0$ gilt, ist eine Ersetzung von b durch sich selbst gleichwertig, ansonsten abschätzend nach oben.

```
        /* Fall 1: leerer Basisblock, hat genau einen Nachfolger, */
2                       /* ist Funktionseinstieg */

4 ...
  // Vor der Optimierung:
6 int ffNumbersOfPredsOfSuccessor = s->GetNumberOfPred();
  ...
8 // Während der Optimierung:
  if ( ffNumbersOfPredsOfSuccessor > 1) // |ps| > 0?
10   flowfactsEstimateBB( b, b );

12 flowfactsExchangeBB( s, b );
  ...
```

Abbildung 5.32: Update für *Empty Basicblock Elimination*, Fall 1

Abbildung 5.32 zeigt die notwendigen Aufrufe der Updatemechanismen.

Fall 2: Eindeutiger Nachfolger & nicht Funktionseinstieg

Der leere Basisblock b ist nicht der Funktionseinstieg und hat genau einen Nachfolger. Daher kann der Kontrollfluss von den Vorgängern von b direkt zu dessen Nachfolger übergehen. Betroffen sind Flow Facts, die sich auf die Basisblöcke b und s beziehen.

Vor der Optimierung gilt:

$$\#(s) = \#(b \rightarrow s) + \sum_{k \in K} \#(ps_k \rightarrow s) \qquad \text{s. Abb. 5.31} \qquad (5.26)$$

$$\#(b) = \sum_{i \in I} \#(p_i \rightarrow b) \qquad \text{s. Abb. 5.31} \qquad (5.27)$$

$$\#(b) = \#(b \rightarrow s) \qquad \text{da } |succ(b)| = 1 \qquad (5.28)$$

Aus (5.26) - (5.28) folgt:

$$\#(s) = \sum_{i \in I} \#(p_i \rightarrow b) + \sum_{k \in K} \#(ps_k \rightarrow s) \qquad (5.29)$$

Durch die Optimierung ist sichergestellt, dass folgende Beziehung gilt:

$$\sum_{i \in I} \#(p_i \rightarrow b) = \sum_{i \in I} \#(p_i \rightarrow s) \qquad (5.30)$$

Nach der Optimierung gilt:

$$\#(s) = \sum_{i \in I} \#(p_i \rightarrow s) + \sum_{k \in K} \#(ps_k \rightarrow s) \qquad \text{s. Abb. 5.31} \qquad (5.31)$$

Da nach (5.30) die Ausführungshäufigkeit von s vor und nach der Optimierung ((5.29) bzw. (5.31)) gleich ist, ist kein Update für s notwendig.

```
        /* Fall 2: leerer Basisblock, hat genau einen Nachfolger, */
2                       /* ist nicht Funktionseinstieg */

4 ...
  // Vor der Optimierung:
6 int ffNumbersOfPredsOfSuccessor = s->GetNumberOfPred();
  ...
8 // Während der Optimierung:
  if ( ffNumbersOfPredsOfSuccessor > 1) // |ps| > 0?
10   flowfactsEstimateBB( b, s );
  else
12   flowfactsExchangeBB( b, s );
  ...
```

Abbildung 5.33: Update für *Empty Basicblock Elimination*, Fall 2

Die Ausführungshäufigkeit von s ist nach der Optimierung gemäß (5.27) und (5.31) um $\sum_{k \in K} \#(ps_k \rightarrow s)$ größer als die von b vor der Optimierung. Nur wenn $|ps| = 0$ gilt, wird b durch s gleichwertig ersetzt. Für $|ps| > 0$, ist s eine abschätzende Ersetzung nach oben.

Abbildung 5.33 zeigt die notwendigen Aufrufe der Updatemechanismen.

Fall 3: Weder Vorgänger noch Nachfolger, nicht Funktionseinstieg

Basisblock b ist nicht der Funktionseinstieg und hat weder Vorgänger noch Nachfolger. Dementsprechend wird b im Kontrollfluss nicht erreicht und kann gelöscht werden (sofern er nicht der letzte Basisblock einer Funktion ist). Betroffen sind Flow Facts, die sich auf den Basisblock b beziehen.

```
      /* Fall 3: leerer Basisblock, hat weder Vorgänger noch Nachfolger */
2   /* ist weder letzter Basisblock einer Funktion noch Funktionseinstieg */

4 ...
  // Während der Optimierung:
6 flowfactsRemoveUnreachableBB( b );
  ...
```

Abbildung 5.34: Update für *Empty Basicblock Elimination*, Fall 3

Vor der Optimierung gilt:

$$\#(b) = 0 \qquad \text{Fehlende Vorgänger} \qquad (5.32)$$

Basisblock b lässt sich gemäß Gleichung (5.32) gleichwertig durch Null ersetzen.

Abbildung 5.34 auf der vorherigen Seite zeigt die notwendigen Aufrufe der Updatemechanismen.

6 Evaluation

Die in den Kapiteln 4 und 5 vorgestellten Modelle und Mechanismen zur Unterstützung von Flow Facts in einem WCET-optimierenden Compiler wurden im Rahmen der Diplomarbeit für den WCC implementiert. Um die Mechanismen testen zu können, wurden die Optimierungstechniken von ICD-C und ICD-LLIR auf ihre Auswirkungen auf Flow Facts – wie ein Beispiel in Abschnitt 5.8.3 zeigt – analysiert und ggf. durch Aufrufe der entsprechenden Updatemechanismen erweitert. Einen Überblick, welche Optimierung durch welche Updatemechanismen zu unterstützen ist, gewährt Anhang A auf Seite 113.

Eine Testreihe auf dieser Implementierung demonstriert die Funktionsfähigkeit der vorgestellten Modellierungen und Transformationen. Dazu wird in Abschnitt 6.1 ein Testverfahren motiviert und entwickelt, dessen Ergebnisse in Abschnitt 6.2 analysiert werden.

6.1 Vergleichtest

Zur Entwicklung eines Tests sind die Benutzer-Schnittstellen für Flow Fact Spezifikationen im WCC vor der Diplomarbeit und nach der Diplomarbeit zu vergleichen:

- **Flow Facts durch Low Level Annotierung (LLA)**

  Auch vor der Diplomarbeit wurden bereits Flow Facts – allerdings nicht von anderen Informationen für die WCET-Analyse wie z. B. Speicheradressbereiche unterschieden – unterstützt. Ihre Annotation war in der Konfigurationsdatei wccrc des WCCs möglich. Dabei mussten sich die Flow Facts auf die Struktur des Programms nach Code Selection und Optimierung durch den Compiler, also auf eine Low Level Darstellung des Programms, beziehen.

- **Flow Facts durch High Level Annotierung (HLA)**

  Durch die Unterstützung von Flow Facts in allen Abstraktionsebenen des Compilers sowie durch die Translations- und Updatemechanismen für Flow Facts – notwendig durch Änderungen der Programmdarstellung im Compiler – wird es einem Programmierer erlaubt, den Quellcode – also eine High Level Darstellung seines Programms – mit Flow Facts zu annotieren. Dies ist insbesondere daher günstig, weil die Flow Facts i. d. R. auf dieser (oder einer vergleichbaren) Abstraktionsebene ermittelt werden.

Da der WCC auch vor der Diplomarbeit die Möglichkeit bot, Flow Facts zu spezifizieren, um die Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ zu ermöglichen, ist die Durchführung eines vergleichenden Tests zwischen der bisherigen Annotation von Flow Fact und dem neu entwickelten Mechanismus derer Annotation nahe liegend. Idealer Weise müssen die Ergebnisse beider Mechanismen

gleich sein, da letztendlich semantisch äquivalente Informationen an das Tool aiT übergeben werden sollten.

Neben der Spezifikationsebene für Flow Facts ist bei den Tests ein weiterer wesentlicher Unterschied festzuhalten. Während die Flow Facts aus der HLA automatisch für die Verwendung durch aiT transformiert werden, sind diese Transformationen bei einer LLA „von Hand“ durchzuführen. Da aber der WCC vier verschiedene Optimierungsstufen kennt, nämlich O0 bis O3 – die Möglichkeit frei wählbarer Kombinationen von Optimierungen sei hier nicht berücksichtigt –, sind für jede dieser Stufen individuelle LLAs notwendig, während eine HLA genügt.

Die Flow Facts der HLA entsprechen im Wesentlichen denen der LLA bei O0.

Beispiel für Annotationen

Die Annotationen für die verschiedenen Testbestandteile werden für den Test *Matmult* vorgestellt. Der Test Matmult ist Bestandteil eines *Benchmarks* für WCET-Analyse-Tools vom *Mälardalen Real-Time Research Center* [Mälardalen Real-Time Research Center 2006]. Er implementiert die Multiplikation zweier 20x20 Matrizen, nutzt multiple Aufrufe einer Funktion, verschachtelte Funktionsaufrufe und dreifach verschachtelte Schleifen.

```
# Annotation für O0
LOCAL : Initialize ,1 ,21 ,21
LOCAL : Initialize ,2 ,21 ,21
LOCAL : Multiply ,1 ,21 ,21
LOCAL : Multiply ,2 ,21 ,21
LOCAL : Multiply ,3 ,21 ,21
```

```
# Annotation für O1
LOCAL : Initialize ,1 ,21 ,21
LOCAL : Initialize ,2 ,21 ,21
LOCAL : Multiply ,1 ,21 ,21
LOCAL : Multiply ,2 ,21 ,21
LOCAL : Multiply ,3 ,21 ,21
```

```
# Annotation für O2
LOCAL : Initialize ,1 ,400 ,400
LOCAL : Multiply ,1 ,20 ,20
LOCAL : Multiply ,2 ,20 ,20
LOCAL : Multiply ,3 ,20 ,20

```

```
# Annotation für O3
LOCAL : main ,1 ,200 ,200
LOCAL : main ,2 ,200 ,200
LOCAL : Multiply ,1 ,20 ,20
LOCAL : Multiply ,2 ,20 ,20
LOCAL : Multiply ,3 ,10 ,10
```

Abbildung 6.1: Flow Facts per LLA für Matmult

Die für die LLA benötigten Optimierungsstufen-spezifischen Annotationen werden in Abb. 6.1 vorgestellt. Für O0 werden alle fünf Schleifen des Programms mit genau 20 Iterationen annotiert (d. h. je 21 Prüfungen der Abbruchbedingungen, da *for*-Schleifen genutzt werden). Durch O1 gibt es keine die Flow Facts betreffenden Änderung, aber durch O2 werden zum Einen die beiden Schleifen in Initialize durch die Optimierung *Loop Collapsing* zu einer Schleife vereint (mit 400 Iterationen, also 401 Abbruchbedingungsprüfungen), zum Anderen werden die Schleifen durch die Optimierung *Transform Head Controlled Loops* in *do-while*-Schleifen transformiert (deren Abbruchbedingung nur 20 bzw. 400 Prüfungen unterliegt).

Durch O3 wird nach dem *Loop Collapsing* (s. o.) durch die *Inline Expansion* je eine Kopie der Initialize-Methode für ihre beiden Aufrufe in main eingefügt. Anschließend werden alle nunmehr drei Vorkommen der Schleife durch ein *Loop Unrolling* um Faktor zwei abgerollt,

deren Iterationshäufigkeit wird dementsprechend auf 200 Iterationen halbiert (201 Prüfungen der Abbruchbedingung). Dieselbe Optimierung wird für die dritte Schleife in der Multiply-Methode durchgeführt. Da die Methode Initialize nach Inlining nicht mehr aufgerufen wird, wird sie durch die Optimierung *Remove Unused Symbols* entfernt. Abschließend erfolgt wieder die Transformation durch die Optimierung *Transform Head Controlled Loops* wie in O2.

```
  # Auszug aus Matmult
2 void Initialize( matrix Array )
  {
4   int OuterIndex, InnerIndex;

6   _Pragma("loopbound min 20 max 20")
    for ( OuterIndex = 0; OuterIndex < UPPERLIMIT; OuterIndex++ ) {
8
      _Pragma("loopbound min 20 max 20")
10    for ( InnerIndex = 0; InnerIndex < UPPERLIMIT; InnerIndex++ )
        Array[OuterIndex][InnerIndex] = RandomInteger();
12
    }
14 }
```

Abbildung 6.2: Flow Facts per HLA für Matmult

Für die HLA wird dagegen nur eine Annotierung im Quellcode benötigt (ein Auszug ist in Abb. 6.2 dargestellt). Durch die Updatemechanismen werden letztendlich die selben Informationen wie durch die LLA für jede der Optimierungsstufen bereitgestellt, wie die entsprechenden Auszüge aus den Protokolldateien für Flow Facts in Abb. 6.3 auf der nächsten Seite zeigen.

Ablauf des Vergleichstests

Der Vergleichstest besteht aus insgesamt drei Schritten und wird für jeden Testfall jeweils für jede der Optimierungsstufen O0 - O3 ausgeführt:

1. Compilieren mit Flow Facts per LLA (alter Mechanismus)

   Dieser Test benötigt für jede Optimierungsstufe eigene Annotationen per wccrc-Datei. Für die vier verschiedenen Optimierungsstufen wird daher je eine eigene Flow Fact Spezifikation verwaltet. Die Ergebnisse dienen als Vergleichswert.

2. Compilieren mit Flow Facts per HLA (neuer Mechanismus)

   Für diesen Test ist nur eine annotierte Quellcode-Datei für alle Optimierungsstufen zusammen notwendig. Die Flow Facts wurden analog zu denen im alten Mechanismus bei Optimierungsstufe O0 annotiert.

3. Compilieren mit Flow Facts per HLA ohne Loopbounds

   Da im vorangegangenen Test primär Loopbounds zum Einsatz kamen, wird ein zweiter Test durchgeführt, bei dem nur Flowrestrictions zur Annotation der Flow Facts genutzt

```
# Annotation für O0, O1 analog
********** List of Flow Facts **********
Loopbound min: 21 max: 21 for: 0x80ecef8
Loopbound min: 21 max: 21 for: 0x80e9480
Loopbound min: 21 max: 21 for: 0x8136508
Loopbound min: 21 max: 21 for: 0x812a968
Loopbound min: 21 max: 21 for: 0x8126ef0
```

```
# Annotation für O2
********** List of Flow Facts **********
Loopbound min: 20 max: 20 for: 0x8141760
Loopbound min: 20 max: 20 for: 0x8134c48
Loopbound min: 20 max: 20 for: 0x8123c38
Loopbound min: 400 max: 400 for: 0x80ff318
```

```
# Annotation für O3
********** List of Flow Facts **********
Loopbound min: 20 max: 20 for: 0x8128c10
Loopbound min: 20 max: 20 for: 0x810a6b8
Loopbound min: 200 max: 200 for: 0x80ca5a8
Loopbound min: 200 max: 200 for: 0x80d59e8
Loopbound min: 10 max: 10 for: 0x8134428
```

Abbildung 6.3: Protokollauszüge zur HLA für Matmult

werden. Dementsprechend kann u. U. auch eine größere $\mathrm{WCET}_{\mathrm{est}}$ als beim Vergleichstest berechnet werden. Die Ursache ist im Abschnitt 4.5 erläutert worden und führte schließlich zur Erweiterung der Flowrestrictions um Loopbounds.

Dieser Test wird durchgeführt, um zu zeigen, dass insbesondere auch die entwickelten Updatemechanismen für Flowrestrictions erfolgreich im WCC eingesetzt werden.

Selbstverständlich wird für alle Tests dieselbe Konfiguration des Compilers (incl. aiT) genutzt.

Für den Vergleichstest werden Testprogramme (vgl. Tabelle 6.1 ff.) genutzt, die durch Heiko Falk und Paul Lokuciejewski nach ausführlichen Analysen für den bisherigen Mechanismus bereits mit Annotationen versehen wurden. Daher kann für diese Testreihe davon ausgegangen werden, dass diese Annotationen fehlerfrei und präzise sind.

6.2 Ergebnisse des Vergleichtests

In den vier Tabellen 6.1 - 6.4 werden die Ergebnisse des Vergleichstest – wie im vorangegangenen Abschnitt beschrieben – für die vier Optimierungsstufen O0 bis O3 dargestellt. In der Spalte *Flow Facts per LLA* sind die Ergebnisse der WCET-Analyse mit den Mechanismen vor der Diplomarbeit als Kontrollwert vermerkt, in der Spalte *Flow Facts per HLA* dagegen die

Vergleichstest bei O0					
Testprogramm	Flow Facts per LLA	Flow Facts per HLA		Flow Facts per HLA (ohne Loopbounds)	
	WCET	WCET	Überab.	WCET	Überab.
binarysearch	325	325	✓	325	✓
countnnegative	101.444	101.444	✓	101.444	✓
cover	38.918	38.918	✓	72.436	86,12%
crc	153.930	153.930	✓	154.014	0,05%
expint	1.041.615	1.041.615	✓	1.041.615	✓
fdct	85.513	85.513	✓	85.513	✓
fibcall	612	612	✓	657	7,35%
fir	332.102	332.102	✓	332.102	✓
insertsort	3.636	3.636	✓	3.636	✓
janne_complex	428	428	✓	428	✓
jfdctint	84.107	84.107	✓	84.107	✓
lcdnum	1.529	1.529	✓	1.529	✓
matmult	8.499.487	8.499.487	✓	8.499.487	✓
petrinet	11.258	11.258	✓	11.258	✓
recursion	unbounded	4.004	-	4.004	-
searchmultiarray	22.750	22.750	✓	22.750	✓

Tabelle 6.1: Testergebnisse Vergleichstest bei O0

Vergleichstest bei O1					
Testprogramm	Flow Facts per LLA	Flow Facts per HLA		Flow Facts per HLA (ohne Loopbounds)	
	WCET	WCET	Überab.	WCET	Überab.
binarysearch	325	325	✓	325	✓
countnnegative	97.024	97.024	✓	97.024	✓
cover	37.567	37.567	✓	70.055	86,48%
crc	150.581	150.581	✓	150.665	0,06%
expint	1.004.816	1.004.816	✓	1.004.816	✓
fdct	101.574	101.574	✓	101.574	✓
fibcall	584	584	✓	629	7,71%
fir	331.562	331.562	✓	331.562	✓
insertsort	3.302	3.302	✓	3.302	✓
janne_complex	427	427	✓	427	✓
jfdctint	52.712	52.712	✓	52.785	0,14%
lcdnum	1.528	1.528	✓	1.528	✓
matmult	8.493.772	8.493.772	✓	8.493.772	✓
petrinet	11.028	11.028	✓	11.028	✓
recursion	unbounded	4.003	-	4.003	-
searchmultiarray	21.334	21.334	✓	21.334	✓

Tabelle 6.2: Testergebnisse Vergleichstest bei O1

Ergebnisse der WCET-Analyse mit den Mechanismen dieser Diplomarbeit, ergänzt um einen Vergleich zum Kontrollwert. Die Spalte *Flow Facts per HLA (ohne Loopbounds)* dokumentiert die Ergebnisse der neuen Mechanismen, wenn auf Loopbounds verzichtet wird, sowie ggf. die prozentuale Abweichung zum jeweiligen Kontrollwert.

Vergleichstest bei O2					
Testprogramm	Flow Facts per LLA	Flow Facts per HLA		Flow Facts per HLA (ohne Loopbounds)	
	WCET	WCET	Überab.	WCET	Überab.
binarysearch	323	323	✓	323	✓
countnnegative	95.587	95.587	✓	95.587	✓
cover	35.772	35.772	✓	66.985	87,23%
crc	136.581	136.581	✓	136.663	0,06%
expint	1.291.037	1.291.037	✓	1.291.037	✓
fdct	135.986	135.986	✓	137.400	1,04%
fibcall	462	462	✓	484	4,76%
fir	331.604	331.604	✓	331.604	✓
insertsort	-	-	-	20.721	-
janne_complex	394	394	✓	394	✓
jfdctint	79.759	79.759	✓	82.467	3,40%
lcdnum	1.526	1.526	✓	1.525	-0,07%
matmult	13.430.230	13.430.230	✓	13.430.230	✓
petrinet	11.053	11.053	✓	11.053	✓
recursion	unbounded	4.003	-	4.003	-
searchmultiarray	19.113	19.113	✓	19.112	-0,01%

Tabelle 6.3: Testergebnisse Vergleichstest bei O2

Vergleichstest bei O3					
Testprogramm	Flow Facts per LLA	Flow Facts per HLA		Flow Facts per HLA (ohne Loopbounds)	
	WCET	WCET	Überab.	WCET	Überab.
binarysearch	323	323	✓	323	✓
countnnegative	25.961	25.961	✓	25.961	✓
cover	35.746	35.746	✓	66.959	87,31%
crc	123.707	123.707	✓	123.787	0,06%
expint	1.218.964	1.218.964	✓	1.218.964	✓
fdct	135.986	135.986	✓	137.400	1,04%
fibcall	487	487	✓	509	4,52%
fir	331.604	331.604	✓	331.604	✓
insertsort	-	-	-	20.721	-
janne_complex	391	391	✓	392	0,26%
jfdctint	79.568	79.568	✓	82.004	3,06%
lcdnum	1.522	1.522	✓	1.522	✓
matmult	14.351.313	14.351.313	✓	14.351.312	-0,00%
petrinet	11.053	11.053	✓	11.053	✓
recursion	unbounded	3.954	-	3.954	-
searchmultiarray	19.113	19.113	✓	19.112	-0,01%

Tabelle 6.4: Testergebnisse Vergleichstest bei O3

Auswertung LLA – HLA

Zur Bewertung der vorgestellten Modelle und Algorithmen für Flow Facts ist insbesondere der Vergleich von LLA und HLA interessant. Hier wird tatsächlich in allen Testfällen (mit Vergleichsmöglichkeit) exakt die gleiche $\mathrm{WCET}_{\mathrm{est}}$ ermittelt. Dieser Vergleich belegt, dass der in dieser Diplomarbeit gewählte Ansatz zur Lösung der Problemstellung aus Abschnitt 3.2

geeignet ist.

Während der Vergleichstest für die Optimierungsstufe O0 zeigt, dass die Flow Facts aus dem Quellcode der Testprogramme korrekt durch die Translationsmechanismen für eine Nutzung durch aiT aufbereitet wurden, zeigen die Optimierungsstufen O1 - O3, dass auch bei Durchführung verschiedener Optimierungen des Quellprogramms alle Flow Facts durch die Updatemechanismen aus ihrer Darstellung für O0 korrekt für aiTs WCET-Analyse bereitgestellt werden. Weder die Aufbereitung von Kontrollflussinformationen für ihre Darstellung mit Bezug auf eine Low Level Intermediate Representation noch deren Transformation für die verschiedenen Optimierungsstufen müssen länger vom Programmierer übernommen werden.

Auffällig ist jedoch, dass für das Testprogramm insertsort für die Optimierungsstufen O2 und O3 weder für die LLA noch für die HLA Ergebnisse vorliegen. Dies liegt an der Nutzung einer frühen *LLIR2CRL* Konverter-Version in der Testimplementierung. Ebenso ist auch die Spezifikation einer Rekursionstiefe erst in einer neueren Version des Konverters möglich, so dass auch für das Testprogramm recursion noch keine Vergleichswerte vorliegen. Aber immerhin zeigen dessen Ergebnisse mit HLA, dass die Beschränkung einer Rekursion möglich ist, da ohne diese Annotierung die Berechnung der $\mathrm{WCET}_{\mathrm{est}}$ mangels grundlegender Begrenzungsinformationen abgebrochen worden wäre.

Auswertung LLA – HLA (ohne Loopbounds)

Der Test mit HLA ohne Loopbounds liefert – wie in Abschnitt 4.5 begründet – erwartungsgemäß z. T. ungenauere $\mathrm{WCET}_{\mathrm{est}}$-Ergebnisse. Auch ohne Betrachtung des Tests recursion, da dort ein entsprechender Vergleichswert fehlt, weichen mehr als die Hälfte der Tests (60%) in mindestens einer Optimierungsstufe vom Vergleichswert ab, insgesamt sind es immer noch rund 38% der Tests über alle Optimierungsstufen. Hier sind jedoch zwei interessante Beobachtungen zu vermerken.

1. Abweichung ist relativ konstant

   Deutlich von seinem Vergleichswert weicht insbesondere der Test cover ab. Allerdings ist dort schon ohne den Einsatz von Updatemechanismen (also bei Optimierungsstufe O0) eine Abweichung von 86,12% zu beobachten. Interessant ist, dass die Größenordnung der Abweichung über alle Optimierungsstufen relativ konstant bleibt. Im Vergleich zum Wert von O0 schwankt die prozentuale Abweichung nur um 1,10 Prozentpunkte.

   Die größte gemessene Schwankung der prozentualen Abweichung einer WCET-Analyse per HLA ohne Loopbounds zu einer WCET-Analyse per LLA über alle Optimierungsstufen mit 3,40 Prozentpunkten kann beim Test jfdctint beobachtet werden.

   Diese relativ geringen Schwankungen sind ein deutliches Indiz dafür, dass die Updatemechanismen auch für Flowrestrictions präzise arbeiten. Sonst wären deutlich größere Schwankungen zu erwarten.

2. Abweichung um genau einen Taktzyklus

   Für vier Testprogramme kann bei der Abweichung der Testergebnisse der HLA ohne Loopbounds zum Vergleichswert durch die LLA nur ein Unterschied von genau einem Taktzyklus ausgemacht werden (janne_complex, lcdnum, matmult und searchmultiarray). Da aber – wenn durch eine unpräzise Annotation auch nur eine Iteration einer Schleife

mehr oder weniger erlaubt werden würde – die Abweichung größer als eins sein müsste, scheint diese Abweichung nur durch eine unterschiedliche Annotierung gleicher Informationen für aiT – es werden dessen Flow Constaints statt dessen Loop Bounds genutzt – zu entstehen.

Da aber in den meisten Fällen eine Abweichung nach unten zu beobachten ist und eine Verletzung der Sicherheit der $WCET_{est}$-Berechnung durch einen Updatemechanismus ausgeschlossen werden muss, ist zu verifizieren, dass keine Flowrestriction den Kontrollfluss fälschlicherweise zu stark einschränkt.

Für diesen Nachweis wurden die vier Tests sowohl mit Flowrestrictions als auch mit Loopbounds annotiert. Da jeweils das Ergebnis des entsprechenden Tests nur mit Loopbounds (HLA) rekonstruiert werden konnte (und in der Berechnung stets die restriktivste Information entscheidend ist), können die Flowrestrictions den Kontrollfluss nicht zu stark einschränken, so dass die Sicherheit gewährt ist.

Fazit

Die Vergleichstests zeigen, dass die Modellierungen und Transformationsmechanismen dieser Diplomarbeit tatsächlich zur Unterstützung einer WCET-Analyse mit Annotierung von Flow Facts auf hoher Abstraktionsebene geeignet sind. Sie zeigen aber auch, dass die Ergänzung der Flowrestrictions durch Loopbounds – wie in Abschnitt 4.5 begründet – notwendig für möglichst präzise WCET-Analyse war.

7 Zusammenfassung und Ausblick

Aufgabe der Diplomarbeit war, Flow Facts für einen WCET-optimierenden Compiler wie den WCC zu entwickeln und in allen Abstraktionsebenen des Compilers zu unterstützen. Die wichtigsten Ergebnisse dieser Arbeit werden in Abschnitt 7.1 kurz zusammengefasst. Abschnitt 7.2 gewährt einen Überblick über Erweiterungsmöglichkeiten für die entworfenen Verfahren.

Ein Ausblick in Abschnitt 7.3 auf die aktuellen Entwicklungen und Forschungen für den WCC – auch mit Blick, inwiefern sich diese Arbeit dort eingliedert und zukünftige Entwicklungen unterstützt bzw. ermöglicht – schließt dieses Dokument.

7.1 Ergebnisse

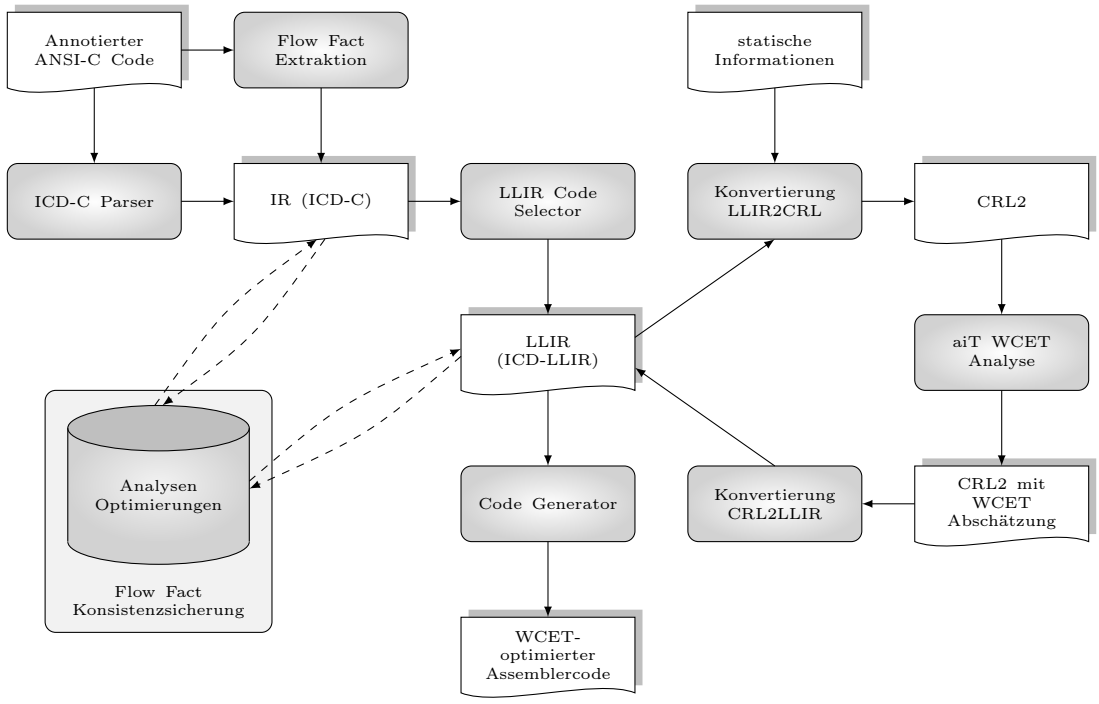


Abbildung 7.1: WCC: aktueller Aufbau

Die Spezifizierung von Kontrollflussinformationen für eine WCET-Analyse durch das Tool aiT der Firma AbsInt im WCC war, statt auf niedriger Abstraktionsebene in einer Konfigurationsdatei, auf einer hohen Abstraktionsebene, nämlich im Quellcode zu ermöglichen. Dazu waren

zum Einen diese Kontrollflussinformationen – im Rahmen dieser Arbeit Flow Facts genannt – zu modellieren, zum Anderen Transformationsmechanismen zu deren Konsistenzsicherung zu entwickeln.

Nachdem bereits in den ersten Kapiteln nicht nur die WCET-Analyse und der WCC selbst vorgestellt worden waren, sondern auch insbesondere die Informationslücke bzgl. der $\mathrm{WCET}_{\mathrm{est}}$-Berechnung analysiert wurde (Stichwort: u. a. grundlegende Begrenzungsinformationen), wurde in Kapitel 4 die Modellierung von Flow Facts als Flowrestrictions und Loopbounds begründet und dargestellt. Zudem wurde ein mathematisches Modell für diese entwickelt. Datenstrukturen für Flow Facts in den verschiedenen Intermediate Representations des WCCs wurden entworfen und inkl. Verwaltungsmechanismen für effiziente Zugriffe implementiert.

In Kapitel 5 wurde veranschaulicht, welche Transformationen für Flow Facts notwendig sind. Mechanismen zur Translation zwischen den Abstraktionsebenen und für Updates innerhalb einer Abstraktionsebene wurden erläutert. Insbesondere für die Updatemechanismen, die i. d. R. durch Optimierungen genutzt werden, wurde ein flexibles Konzept geschaffen, das auch die Unterstützung neuer Optimierungen bei geringem Aufwand erlaubt. Bei allen Transformationen wurde insbesondere die Sicherheit aber auch die Präzision dieser bzgl. der WCET-Analyse bedacht und so weit möglich erörtert.

Um die Qualität der entwickelten Modelle und Mechanismen zu demonstrieren, waren diese in den WCC zu integrieren. Dazu wurden die Veränderungen eines Quellprogramms im WCC, insbesondere durch dessen Optimierungen, analysiert und durch die entwickelten Transformationen unterstützt. Die Auswertung eines (erfolgreichen) Vergleichstests wurde in Kapitel 6 durchgeführt.

Die Struktur des WCCs zum Abschluss dieser Arbeit wird in Abb. 7.1 auf der vorherigen Seite dargestellt, eine Gegenüberstellung dessen Struktur vor der Diplomarbeit und seiner Änderungen in der Diplomarbeit ist auf den Seiten 24 und 25 in Abschnitt 3.3 zu finden.

7.2 Erweiterungen der Updatemechanismen für Flow Facts

Auch wenn grundlegende Mechanismen zur Unterstützung von Flow Facts im WCC erfolgreich entwickelt wurden, können diese an verschiedenen Stellen um neue Funktionen erweitert werden.

- Explizite Unterstützung der Schleifentransformation von aiT

  In Abschnitt 2.3.1 auf Seite 10 wurde beschrieben, dass aiT alle erkannten Schleifen in eigene Routinen kapselt. Nur diese erkannten Schleifen dürfen durch aiTs Loop Bounds annotiert werden, alle anderen sind durch potentiell weniger präzise Flow Constraints zu annotieren. Darüber hinaus dürfen Flow Constraints nur mit dem präziseren Qualifier EACH genutzt werden, wenn sie nach der Schleifentransformation noch vollständig zu einer Routine gehören.

  Ein Abgleich zwischen CRL2 und der LLIR unmittelbar nach der Schleifentransformation von aiT würde erlauben, alle Flow Facts mit maximaler Genauigkeit für aiT zu

spezifizieren, während z. Z. mangels notwendiger Informationen z. B. alle Flowrestrictions als Flow Constraints mit dem Qualifier SUM annotiert werden müssen.

- Erweiterung der Suche nach gleichwertigen Ersetzungen auf Statementebene

  Die Interpretation der Definition von gleichwertigen Entscheidungsvariablen in Abschnitt 5.2.2 auf Seite 49 verbietet z. B., nach einer gleichwertigen Ersetzung über die eigene Schleifenebene hinaus zu suchen, um zu gewährleisten, dass das Update der Flow Facts auch für die Analyse durch aiT volle Ausdruckskraft behält. Eine genaue Analyse, wann für aiT Ersetzungen von Statements in ICD-C gleichwertig sind, mag die Menge möglicher Ersetzungen über die eigene Schleife hinaus vergrößern.

- Erweiterung der Suche nach abschätzenden Ersetzungen auf Basisblockebene

  Bisher wird die Suche nach abschätzenden Ersetzungen auf eine Funktion begrenzt, indem eine Suche, wenn ein Funktionseinstieg erreicht wird, über dessen Vorgänger, oder wenn ein Funktionsausstieg erreicht wird, über dessen Nachfolger verboten wird. Hier ist vorstellbar, über die Grenzen einer Funktion hinwegzusuchen.

- ...

7.3 Zukünftige Entwicklung des WCCs

Ziel des WCCs ist die Unterstützung der Forschung bei der Entwicklung von WCET-reduzierenden Optimierungen in einem Compiler. Dies kann nur gelingen, wenn zum Einen die $\mathrm{WCET}_{\mathrm{est}}$ automatisch ermittelt und für eine Optimierung genutzt werden kann, und wenn zum Anderen eine durchgeführte Optimierung Flow Facts aktualisieren und annotieren kann, um eine Veränderung der $\mathrm{WCET}_{\mathrm{est}}$ zu beobachten. Letzteres wird auf jeder Abstraktionsebene im WCC durch diese Arbeit ermöglicht.

Darüber hinaus ist die Beurteilung einer Optimierung nur möglich, wenn die $\mathrm{WCET}_{\mathrm{est}}$-Berechnung präzise ist. Wird die $\mathrm{WCET}_{\mathrm{est}}$ um einige hundert Prozent überabgeschätzt, so kann deren Reduktion kein Hinweis auf die Auswirkungen einer Optimierung auf die wirkliche $\mathrm{WCET}_{\mathrm{real}}$ geben und erst recht keine Entscheidungsgrundlage für eine wirtschaftliche Unternehmung darstellen. Die Präzision der Berechnung kann durch automatische Kontrollflussanalysen unterstützt werden, die implizite Flow Facts explizit annotieren, und diese u. U. so für aiTs WCET-Analyse verfügbar machen. Solche Kontrollflussanalysen basieren – wegen der zusätzlich verfügbaren Informationen z. B. durch die Semantik von Statements – i. d. R. auf einer hohen Abstraktionsebene eines Programms. Die in dieser Diplomarbeit entwickelten Mechanismen erlauben auch diesen Analysen, ihre Ergebnisse direkt für die $\mathrm{WCET}_{\mathrm{est}}$-Berechnung zu annotieren.

Durch solche Analysen kann die Anzahl der notwendigen Benutzerannotationen minimiert werden, so dass in Abb. 7.1 auf Seite 109 statt von Flow Fact Extraktion aus dem Quellcode im Idealfall sogar von einer automatisierten Flow Fact Extraktion (ohne Benutzerannotationen) gesprochen werden kann.

A Übersicht Optimierungstechniken

Im Rahmen der Diplomarbeit wird regelmäßig auf Optimierungen von ICD-C und ICD-LLIR verwiesen, da insbesondere wegen dieser Optimierungen Updates von Flow Facts durchgeführt werden. Um einen Überblick über die möglichen Optimierungen zu gewähren, werden in den Abschnitten A.1 und A.2 die entsprechenden Techniken kurz vorgestellt. Zudem wird vermerkt, welche Optimierungen Transformationen von Flow Facts erfordern.

A.1 Optimierungen ICD-C

Die Optimierungen in ICD-C werden in diesem Abschnitt nur kurz beschrieben, ohne auf alle Details (insbesondere alle Vorbedingungen) einzugehen. Weitere Informationen zu den Optimierungen aber auch zu den grundlegenden Konzepten sind z. B. in [ICD-C 2006; Muchnick 1997; Bacon u. a. 1994] zu finden.

In ICD-C sind z. Z. folgende plattformunabhängige Optimierungen verfügbar:

- **Common Subexpression Elimination**

  Statt einen Teilausdruck (eine Expression) wiederholte Male zu berechnen, wird die Berechnung nur einmalig durchgeführt. Deren Ergebnis wird zwischengespeichert und anstelle einer Neuberechnung des Teilausdrucks genutzt.

  Da die Optimierungen auf Ebene der Expressions durchgeführt werden, sind die Flow Facts (mit Bezug auf Statements) nicht betroffen.

- **Fold Constant Code**

  Ein Ausdruck nur mit Konstanten wird zur Compile-Zeit ausgewertet und durch sein Ergebnis ersetzt. Statt in Ausdrücken auf Variablen mit konstantem Wert zuzugreifen, werden diese durch ihren Wert ersetzt. Konstante Bedingungen in Selection oder Loop Statements werden ausgewertet, um diese Ausdrücke zu vereinfachen. So wird z. B. bei einem *if-else*-Statement mit konstant falscher Bedingung das *if-else*-Statement durch seinen *else*-Anweisungsblock ersetzt.

  Durch das Ersetzen von Selection und Loop Statements mit konstanten Bedingungen sind die Flow Facts betroffen. Insbesondere das *Ersetzen eines Statements* wird für deren Aktualisierung genutzt, aber auch das *Löschen von Flow Facts* wird z. B. für die nicht erreichbaren Statements eines Selection Statements benötigt. Spezielle Updatemechanismen für Schleifen sind nicht notwendig, da eine Optimierung von Schleifen nur durchgeführt wird, wenn deren Bedingung konstant falsch ist. Dann aber genügt das *Löschen von Flow Facts*, um Loopbounds vor dem Löschen von Schleifen zu entfernen.

- **Dead Code Elimination**

  Statements, die durch den Kontrollfluss nicht erreicht werden, werden entfernt. Basisblöcke, die unmittelbar aufeinander folgen (der Vorgänger im Kontrollfluss hat genau einen Nachfolger, der Nachfolger genau einen Vorgänger), werden zusammengefasst. Statements ohne Effekt sowie Zuweisungen, deren Ergebnis nie genutzt wird, werden entfernt.

  Durch eine *Dead Code Elimination* sind natürlich auch Flow Facts betroffen. Aber nicht nur das *Löschen von Flow Facts* wird genutzt, um nie ausgeführte Statements korrekt aus Flow Facts zu entfernen. Insbesondere für die Zusammenführung aufeinanderfolgender Basisblöcke kann das *Ersetzen eines Statements* zur Aktualisierung von Flow Facts für die Entfernung von Jump und Targeted Statements genutzt werden. Da die Optimierung dabei den Kontrollfluss analysiert, kann ohne zusätzlichen Rechenaufwand auf die flowfactsAddSingle()-Methode für Basisblöcke zurückgegriffen werden. Aber auch für das Entfernen von Statements ohne Effekt wird das *Ersetzen eines Statements* genutzt.

- **Tail Recursion Elimination**

  Die Rekursion einer Funktionen, die den rekursiven Aufruf stets als letztes ausführt, wird durch einen Sprung an deren Anfang ersetzt.

  Die betroffenen Flow Facts (ausschließlich Flowrestrictions) werden durch den Mechanismus *Ersetzen eines Statements* aktualisiert.

- **Inline Expansion**

  Der Aufruf einer Funktion wird durch eine Kopie des Funktionsrumpfs ersetzt, um den Overhead durch den Funktionsaufruf zu eliminieren. Die Auswahl der Funktionen für diese Optimierung beschränkt sich i. d. R. auf Funktionen mit wenigen Statements und solche, die explizit durch das Schlüsselwort inline gekennzeichnet wurden, da durch die Kopie der Funktionsrümpfe der benötigte Speicherplatz wächst.

  Neben dem *Ersetzen eines Statements* wird insbesondere für die Kopie des Funktionsrumpfs der Updatemechanismus *Anpassen von Flow Facts nach der Kopie eines Statements* genutzt.

- **Function Specialization**

  Enthält der Funktionsaufruf einer Funktion (mit wenigen Statements) ein konstantes Argument, so wird eine spezialisierte Version dieser Funktion erzeugt, in der dieser Parameter entfällt. Dessen Vorkommen im Funktionsrumpf wird durch den konstanten Wert ersetzt.

  Da Kopien von Funktionen angelegt werden, ist der Mechanismus *Anpassen von Flow Facts nach der Kopie eines Statements* zur Aktualisierung der Flow Facts zu nutzen.

- **Create Multiple Exits**

  Nutzen verschiedene Kontrollflussverzweigungen einer Funktion dasselbe *return*-Statement, so wird für jede Verzweigung ein eigenes *return*-Statement angelegt, um unnötige Kontrollflusssprünge zu vermeiden.

  Da hier die Aufgabe eines *return*-Statements nun durch eine Menge von *return*-Statements wahrgenommen wird, sind diese auch in Flowrestrictions durch das *Ersetzen eines Statements* gemeinsam zu vermerken.

- **Eliminate Return Value**

  Falls der Rückgabewert einer Funktion nach keinem Aufruf der Funktion genutzt wird, so wird er aus der Funktion entfernt.

  Diese Optimierung entfernt vor allem Expressions aus *return*-Statements und erfordert daher kein Update von Flow Facts.

- **Separate Life Ranges**

  Jede lokale Variable, die für verschiedene Zwecke mit verschiedenen Lebensbereichen genutzt wird, wird durch je eine lokale Variable für jeden dieser Bereiche ersetzt, um die Flexibilität für andere Optimierungen und für die Registerallokation zu erhöhen.

  Auch diese Optimierung verursacht keine Aktualisierungen für Flow Facts, da lediglich Symboltabellen und Expressions modifiziert werden.

- **Transform Head Controlled Loops**

  Schleifen mit Prüfung ihrer Abbruchbedingung zu Beginn einer jeden Iteration (*for*- und *while*-Schleifen) werden durch eine Schleife mit Prüfung dieser Bedingung zum Ende einer jeden Iteration (*do-while*-Schleife) ersetzt, um unnötige Kontrollflusssprünge zu vermeiden.

  Da die Iterationshäufigkeit einer Schleife durch diese Optimierung nicht modifiziert wird, lassen sich Loopbounds durch den Mechanismus *Versetzen einer Loopbound* aktualisieren. Für Flowrestrictions wird das *Ersetzen eines Statements* genutzt.

- **Loop De-Indexing**

  Wird in einer Schleife durch Änderung einer Schleifenvariable sequentiell auf die Elemente eines Arrays zugegriffen, so wird der Overhead der Adressberechnung für den Zugriff auf ein Element (jeweils Basisadresse + aktueller Offset) durch Nutzung eines direkt in- und dekrementierbaren Zugriffs per Zeiger reduziert.

  Flow Facts sind durch diese Optimierung nicht betroffen.

- **Loop Collapsing**

  Verschachtelte Schleifen zur Iteration mehrdimensionaler Arrays werden zu einer Schleife zusammengefasst, durch die das mehrdimensionale Array analog zu einem eindimensionalen Array iterativ durchlaufen wird.

  Flowrestrictions werden durch das *Ersetzen eines Statements* (in Form des *direkten Ersetzens*) aktualisiert, Loopbounds der alten Schleifen werden durch das *Löschen von Flow Facts* entfernt, die neue Schleife wird durch *Erzeugen einer Loopbound* direkt durch die Analyseergebnisse des Loop Analyzers annotiert.

- **Loop Unswitching**

  Enthält ein Loop Statement nur ein Selection Statement, dessen Bedingung von der Schleife unabhängig ist, so werden Loop Statement und Selection Statement vertauscht (ggf. sind von der Schleife mehrere Kopien zu erzeugen), um unnötige mehrfache Tests der Bedingungen von Loop und Selection Statement sowie unnötige Kontrollflusssprünge zu vermeiden.

  Betroffen sind nur Flowrestrictions, die durch das *Ersetzen eines Statements* aktualisiert werden können.

- **Loop Unrolling**

  Um den Overhead der durch Schleifen verursachten Sprünge zu reduzieren, wird die Schleifeniterationshäufigkeit um einen Faktor u reduziert, der Schleifenkörper dafür aber $u-1$ mal kopiert und in die Schleife eingefügt. So wird in Summe die Ausführungshäufigkeit jedes Statements nicht verändert.

  Ein Großteil der Mechanismen wird zur Aktualisierung der Flow Facts eingesetzt: *Ersetzen eines Statements*, *Erzeugen einer Loopbound*, *Ermittlung der Schleifeniterationshäufigkeit*, *Löschen von Flow Facts* sowie das *Anpassen von Flow Facts nach der Kopie eines Statements*.

- **Redundant Load Elimination**

  Um mehrere Lesezugriffe auf eine globale Variable aus einer Funktion heraus zu vermeiden, werden mehrfach genutzte globale Variablen in eine lokale Variable kopiert.

  Da nur Expressions und Symboltabellen modifiziert und höchstens Expression Statements zum Laden der globalen Variablen eingefügt werden, sind Flow Facts durch diese Optimierung nicht betroffen.

- **Expression Simplifikation**

  Enthalten Ausdrücke (Expressions) z. B. redundante Casts oder sich kompensierende Vorkommen von „*“- und „&“-Operatoren, so werden diese Ausdrücke vereinfacht. Präinkrement und -dekrement werden Postinkrements und -dekrements vorgezogen.

  Durch die Vereinfachung von Expressions werden Flow Facts nicht betroffen.

- **Struct Scalarization**

  Lokale *struct*-Objekte, die Bitfelder ausschließlich mit lokalen Zugriffen enthalten, werden in äquivalente Objekte des Typs *Integer* umgewandelt. Skalare Komponenten eines *struct*-Objekts nur mit direkten Zugriffen werden zu eigenständigen Komponenten transformiert.

  Ähnlich wie in der *Redundant Load Elimination* werden Expressions und Symboltabellen modifiziert und Expression Statements eingefügt, Flow Facts sind aber wiederum nicht betroffen.

- **Remove Unused Symbols**

  Symbole, die im Programm nicht genutzt werden, werden aus den lokalen und globalen Symboltabellen entfernt. Dabei werden u. U. auch vollständige, nicht genutzte Funktionen entfernt.

  Für eine nie ausgeführte Funktion wird der Mechanismus *Löschen von Flow Facts* zu deren Aktualisierung genutzt.

- **Remove Unused Function Arguments**

  Argumente einer Funktion, die nicht genutzt werden, werden sowohl aus der Funktionsdeklaration als auch aus den Funktionsaufrufen entfernt.

  Flow Facts sind durch diese Optimierung nicht betroffen.

- **Merge String Constant Expressions**

  Vorkommen identischer konstanter String-Ausdrücke werden zusammengefügt.

Flow Facts sind durch diese Optimierung ebenfalls nicht betroffen.

- **Value Propagation**

  Ausdrücke werden, so weit möglich, an den Punkt ihres Einsatzes verschoben. Dies betrifft insbesondere konstante Integer-Ausdrücke, aber z. B. auch Symbol-Ausdrücke mit einmaliger Definition.

  Da die Optimierung auf Ebene von Expressions arbeitet, sind Flow Facts nicht betroffen.

Die Abfolge der Optimierungen ist bezüglich der Aktualisierung von Flow Facts nicht wesentlich. Sie wird in [ICD-C 2006] vorgestellt.

Ergänzt werden diese Optimierungen um einen *Loop Analyzer*, der für ein Loop Statement von der Ermittlung dessen Bedingung bis hin zur Ermittlung dessen Ausführungshäufigkeit verschiedene Analysen durchführt. Die Ermittlung von Ausführungshäufigkeiten z. B. wird aber nur für einfache Schleifen erfolgreich durchgeführt.

Die Ergebnisse des Loop Analyzers werden genutzt, um u. a. zu entscheiden, ob Optimierungen einer Schleife wie z. B. durch ein *Loop Unrolling* durchzuführen sind. Allerdings bieten die Optimierungen i. d. R. zusätzlich die Möglichkeit, durch eine externe Funktion diese Entscheidungsfindung zu beeinflussen, so dass nicht immer zwingend Analyseergebnisse durch einen Loop Analyzer vorliegen müssen.

A.2 Optimierungen ICD-LLIR

In diesem Abschnitt werden kurz die ICD-LLIR Optimierungen vorgestellt. Details sind in [ICD-LLIR 2006] zu finden. Die Optimierung *Empty Basicblock Elimination* wird – vor allem in Hinblick auf notwendige Updates von Flowrestrictions – detailliert in Abschnitt 5.8.3 analysiert.

In der ICD-LLIR sind z. Z. folgende Optimierungen verfügbar:

- **Empty Basicblock Elimination**

  Basisblöcke, die keine Instruktionen enthalten, werden entfernt.

  Beide Mechanismen – *gleichwertiges Ersetzen eines Basisblocks* und *abschätzendes Ersetzen eines Basisblocks* – werden zur Aktualisierung von Flowrestrictions genutzt, das *Versetzen einer Loopbound* und das *Löschen von Flow Facts* werden für die Aktualisierung von Loopbounds verwendet.

- **Remove Unused Virtual Registers**

  Virtuelle Register, die nicht benutzt werden, werden entfernt.

  Flow Facts sind nicht betroffen.

- **Peephole Optimization**

  Eine kurze Folge von Assemblerbefehlen (*Peephole*) wird betrachtet und mit Beobachtungsmustern verglichen. Wird ein bekanntes Muster entdeckt, wird eine dafür spezifizierte Änderung dieser Assemblerbefehlsfolge durchgeführt.

Flow Facts sind nur betroffen, wenn durch die Änderungen die Ausführungshäufigkeit von Basisblöcken verändert wird oder wenn der Sprung einer Schleife verschoben wird. Bisher sind Flow Facts allerdings nicht betroffen.

- **List Scheduling**

  Die Assemblerbefehle werden umsortiert, um *Pipeline Stalls* infolge von *Hazards* zu vermeiden.

  Da die Umsortierung Daten- und Kontrollflussabhängigkeiten berücksichtigt, sind Flow Facts nicht betroffen.

- **Loop Invariant Code Motion**

  Instruktionen, die von einer Schleife unabhängig sind, werden nach Möglichkeit vor diese Schleife verschoben, um deren Ausführungshäufigkeit zu reduzieren.

  Der Kontrollfluss kann durch diese Optimierung nicht beeinflusst werden. Falls leere Basisblöcke entstehen, werden diese durch die entsprechende Optimierung *Empty Basicblock Elimination* entfernt. Flow Facts sind daher nicht direkt betroffen.

Literaturverzeichnis

AbsInt 2006
AbsInt Angewandte Informatik GmbH (Hrsg.): *Worst-Case Execution Time and Stack Analysis aiT for TriCore.* Version 1.6 r1. AbsInt Angewandte Informatik GmbH, Juli 2006

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

Engblom 1997
Engblom, Jakob: *Worst-case execution time analysis for optimized code*, Uppsala University, Diplomarbeit, September 1997

Engblom u. a. 1998
Engblom, Jakob ; Altenbernd, Peter ; Ermedahl, Andreas: Facilitating Worst-Case Execution Times Analysis for Optimized Code. In: *10th Euromicro Workshop on Real Time Systems* 00 (1998), S. 146–153. – ISSN 1068–3070

Engblom u. Ermedahl 2000
Engblom, Jakob ; Ermedahl, Andreas: Modeling Complex Flows for Worst-Case Execution Time Analysis. In: *21st IEEE Real-Time Systems Symposium*, 2000

Ermedahl 2003
Ermedahl, Andreas: *A Modular Tool Architecture for Worst-Case Execution Time Analysis*, Uppsala University, Diss., Juni 2003

Ermedahl u. a. 2002
Ermedahl, Andreas ; Engblom, Jakob ; Stappert, Friedhelm: A Unified Flow Information Language for WCET Analysis. In: *2nd Intl. Workshop on Worst-Case Execution Time Analysis*, 2002

Falk u. a. 2006
Falk, Heiko ; Lokuciejewski, Paul ; Theiling, Henrik: Design of a WCET-Aware C Compiler. In: Mueller, Frank (Hrsg.): *6th Intl. Workshop on Worst-Case Execution Time Analysis*, Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Deutschland, 2006

Heckmann u. Ferdinand 2004
Heckmann, Reinhold ; Ferdinand, Christian: aiT: Worst-Case Execution Time Prediction by Static Program Analysis. In: *IFIP Congress Topical Sessions*, 2004, S. 377–384

Holsti u. Saarinen 2002
Holsti, Niklas ; Saarinen, Sami: Status of the Bound-T WCET Tool. In: *2nd Intl. Workshop on Worst-Case Execution Time Analysis*, 2002

Infineon 2005a
INFINEON TECHNOLOGIES AG (Hrsg.): *TriCore 1 32-bit Unified Processor Core - Core Architecture*. V 1.3.5. München: Infineon Technologies AG, Februar 2005

Infineon 2005b
INFINEON TECHNOLOGIES AG (Hrsg.): *TriCore 1 32-bit Unified Processor Core - Instruction Set*. V 1.3.5. München: Infineon Technologies AG, Februar 2005

ICD 2007
INFORMATIK CENTRUM DORTMUND (Hrsg.): *Embedded Systems Profit Center (ICD/ES)*. `http://www.icd.de/es`, Abruf: 26. 02. 2007

ICD-CG 2007
INFORMATIK CENTRUM DORTMUND (Hrsg.): *ICD-CG code generator generator*. `http://www.icd.de/es/icd-cg/icd-cg.html`, Abruf: 01. 04. 2007

ICD-C 2006
INFORMATIK CENTRUM DORTMUND (Hrsg.): *ICD-C Compiler framework - Developer Manual*. Informatik Centrum Dortmund, Juni 2006

ICD-LLIR 2006
INFORMATIK CENTRUM DORTMUND (Hrsg.): *ICD Low Level Intermediate Representation backend infrastructure (LLIR) - Developer Manual*. Informatik Centrum Dortmund, Juni 2006

ISO 9899 1999
ISO/IEC (Hrsg.): *Programming languages – C*. 2. Schweiz: ISO/IEC, 1999

Kirner 2000
KIRNER, Raimund: *Integration of Static Runtime Analysis and Program Compilation*. Wien, Österreich, Technische Universität Wien, Institut für Technische Informatik, Diplomarbeit, 2000

Kirner 2002
KIRNER, Raimund: The Programming Language wcetC / Technische Universität Wien, Institut für Technische Informatik. Wien, Österreich, 2002 (2/2002). – Research Report

Kirner 2003
KIRNER, Raimund: *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. Wien, Österreich, Technische Universität Wien, Diss., Mai 2003

Kirner u. a. 2002
KIRNER, Raimund ; LANG, Roland ; FREIBERGER, Gerald ; PUSCHNER, Peter: Fully Automatic Worst-Case Execution Time Analysis for Matlab/Simulink Models. In: *14th Euromicro Conference on Real-Time Systems*, IEEE, Juni 2002, S. 29–36

Kirner u. Puschner 2005a
KIRNER, Raimund ; PUSCHNER, Peter: Classification of Code Annotations and Discussion of Compiler-Support for Worst-Case Execution Time Analysis. In: *5th Intl. Workshop on Worst-Case Execution Time Analysis*, 2005

Kirner u. Puschner 2005b
KIRNER, Raimund ; PUSCHNER, Peter: Classification of WCET Analysis Techniques. In:

8th IEEE Intl. Symposium on Object-oriented Real-time distributed Computing, 2005, S. 190–199

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

Marwedel 2001
MARWEDEL, Peter: *Begleitmaterial zur Vorlesung Rechnergestützter Entwurf / Produktion (Mikroelekronik)*. 2001. – Universität Dortmund

Marwedel 2006
MARWEDEL, Peter: *Embedded System Design*. Dordrecht, Niederlande : Kluwer Academic Publishers, 2006. – ISBN 978–0–387–29237–3

Mälardalen Real-Time Research Center 2006
MÄLARDALEN REAL-TIME RESEARCH CENTER (Hrsg.): *The Worst-Case Execution Time (WCET) analysis project*. Version: September 2006. `http://www.mrtc.mdh.se/projects/wcet/home.html`, Abruf: 13. 04. 2007

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

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

Recht 2004
RECHT, Peter: *Quantitative Methoden: Operations Research I*. 2004. – Begleitmaterial zur Vorlesung, Universität Dortmund

Statistisches Bundesamt 2006
STATISTISCHES BUNDESAMT (Hrsg.): *Tabellenanhang zur Pressebroschüre Informationstechnologie in Haushalten 2005*. 2006

Bound-T 2005
TIDORUM LTD. (Hrsg.): *Bount-T timing analysis tool - User Manual*. Version 3. Tidorum Ltd., Mai 2005

Vasseur 2004
VASSEUR, Clément: Code-generator generators: generating the instruction selector / Laboratoire de Recherche et Développement de l'Epita. 2004 (0403). – Research Report

Vrchoticky 1993
VRCHOTICKY, Alexander: Integrating Compilation and Timing Analysis / Technische Universität Wien, Institut für Technische Informatik. Wien, Österreich, 1993. – Research Report

Welt.de 2006
WELT.DE (Hrsg.): *Autos brauchen immer mehr Elektronik*. Version: Dezember 2006. `http://www.welt.de/data/2006/12/14/1146387`, Abruf: 17. 02. 2007

Stichwortverzeichnis