Statische WCET-Analyse von LLVM Bytecode

Diplomarbeit im Fach Informatik

vorgelegt von

Benjamin Oechslein
geb. am 14.04.1983
in Gunzenhausen (Bayern)

angefertigt am
31. Juli 2008

Department Informatik
Lehrstuhl für Verteilte Systeme und Betriebssysteme (Informatik 4)
Friedrich–Alexander–Universität Erlangen–Nürnberg

Betreuer:	Prof. Dr. W. Schröder-Preikschat
	Dipl.–Inf. Fabian Scheler

Beginn der Arbeit	01. Februar 2008
Abgabe der Arbeit	01. August 2008

Erklärung

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

(Benjamin Oechslein)
Erlangen, den 31. Juli 2008

Kurzfassung

Bei der Entwicklung von Echtzeitsystemen ist die maximal mögliche Ausführungszeit eines Programmstückes, kurz WCET, eine wichtige Kenngröße. Dabei spielt es keine Rolle, ob es sich um ein zeit- oder ereignisgesteuertes System handelt. In beiden Systemtypen wird die WCET der ausgeführten Programme benötigt, um die Korrektheit garantieren zu können. Im Rahmen dieser Arbeit soll eine Möglichkeit vorgestellt werden, die WCET eines Programmes, welches in LLVM-Bytecode vorliegt, statisch zu bestimmen.

Ziel dieser Arbeit ist es, eine Analyse auf Ebene der plattformunabhängigen LLVM-Zwischensprache durchzuführen. Es wird also der Kontrollfluss des Programmes vor der Umwandlung in Assemblercode betrachtet. Das LLVM-Backend wird zur Codeerzeugung für die einzelnen Basisblöcke der LLVM-Zwischensprache verwendet. Zur Bestimmung der maximalen Abarbeitungszeit des Assemblercodes, und damit der einzelnen Basisblöcke, kommt ein externes Werkzeug zum Einsatz. Durch Wechsel des Backends ist es auch möglich, die Analyse auf andere Zielarchitekturen zu portieren. Die eigentliche Bestimmung der WCET findet dann auf einer vom Kontrollflussgraphen des LLVM-Zwischencodes abgeleiteten Datenstruktur statt und verwendet dabei zeitliche Informationen aus dem Backend und direkt aus dem Quellprogramm. Dies umfasst Informationen über Schleifen, rekursive Funktionsaufrufe und die Abarbeitungszeiten der Basisblöcke.

Zu Beginn der Arbeit werden zuerst die Grundlagen der statischen WCET-Analyse erläutert. Darauf folgt eine Beschreibung des LLVM-Frameworks und der Einflüsse, die dessen Design auf die WCET-Analyse haben. Im Anschluss daran wird der Entwurf der Analyse diskutiert und dabei einige zentrale Problemstellungen erläutert. Schließlich folgt eine Beschreibung der Implementierung und deren Evaluation. Die Arbeit schließt mit einem kurzen Ausblick über mögliche Erweiterungen und verwandte Ansätze in anderen Arbeiten.

Abstract

During the development of real-time systems the worst case execution time, WCET, of a given piece of software is a key parameter. It doesn't matter if the system is time- or eventtriggered. The WCET is needed to verify correctness for both types of systems. This paper presents an approach to statically determine the WCET for a program written in LLVM bytecode.

The goal of this paper is to implement a WCET analysis on the abstraction level of the LLVM's platform independent intermediate representation. Therefore the program's control flow is analysed before the translation to low-level assembler code takes place. The LLVM backend does the code transformation to assembly language on a basic block basis. An external tool is used to determine the worst case execution time of the resulting assembly code, which equals the WCET for the basic blocks. Therefore it is possible to adapt this analysis to new target architectures by choosing another LLVM backend for code generation. The actual analysis is performed on a data structure derived from the control flow graph of the LLVM bytecode and uses additional information from the backend and the analysed program. This includes details about loops, recursive function calls and execution times of basic blocks.

The paper starts with a basic explanation of static WCET analysis. Then follows a description of the LLVM framework and its implications on the design of the analysis. The next chapter deals with overall design decisions and focuses on some encountered problems and then describes the concrete implementation. The thesis is then concluded with an evaluation and a short description of possible extensions and related work.

Inhaltsverzeichnis

1 Einleitung

Bei der Entwicklung von Echtzeitsystemen ist es vorrangiges Ziel auf externe Ereignisse rechtzeitig zu reagieren. Je nach Kritikalität des Systems kann eine Verletzung dieser Rechtzeitigkeit schwere Folgen für die Anwendung nach sich ziehen. Für harte Echtzeitsysteme ist es deswegen wichtig, die Einhaltung zeitlicher Schranken zu garantieren. Als Vorraussetzung dafür muss man die maximale Ausführungszeit eines Programmes, kurz WCET genannt, bestimmen. Diese Aufgabe ist jedoch nicht so einfach zu bewerkstelligen, da im Allgemeinen die Ausführungszeit eines Programmes von seinen Eingabedaten abhängt. Eine exakte Bestimmung ist deshalb oft nicht möglich. Auch das Messen der Ausführungszeit gestaltet sich aus diesem Grund schwierig, da man dazu im schlimmsten Fall den kompletten Eingabeparameterraum abdecken müsste, um auch wirklich den Pfad mit der längsten Ausführungszeit zu messen. Für viele Programme lässt sich jedoch eine obere Schranke für die Abarbeitungszeit angeben. Diese lässt sich durch eine statische Analyse des Programmcodes gewinnen.

Besondere Bedeutung kommt der WCET-Analyse im Bereich von komplexeren Echtzeitsystemen zu, die nicht nur einzelne kritische Tasks ausführen, sondern mehrere. Zur Ablaufplanung benötigt man dann die maximale Ausführungszeit jedes einzelnen Tasks, um sicherzustellen, dass alle zeitlichen Anforderungen auch erfüllt werden können. Kann die WCET nicht nach oben hin abgeschätzt werden, so ist es nicht möglich die Erfüllbarkeit aller zeitlichen Vorgaben zu garantieren. Somit ist eine gute WCET-Analyse notwendig, um harte Echtzeitsysteme mit mehreren Tasks zu entwerfen.

1.1 Zeitanalyse für einen Real-Time Systems Compiler

Die Zeitanalyse, die in dieser Arbeit betrachtet wird, soll im Kontext eines *Real-Time System Compiler (RTSC)* [18], [19] Verwendung finden. Dieser beschreibt Echtzeitsysteme als eine Menge von *Atomic Basic Blocks*, kurz ABBs genannt. Ein Echtzeitsystem besteht dabei typischerweise aus mehreren Kontrollflussgraphen, die über verschiedene Arten von Abhängigkeiten miteinander verbunden sind. Dies sind z.B. Stellen im Programmfluss, die nur unter gegenseitem Ausschluss betreten werden dürfen oder Erzeuger-Verbraucher-Beziehungen. Die ein-

zelnen Abschnitte zwischen solchen Verbindungspunkten, an denen die Kontrollflussgraphen interferieren, werden im Kontext des RTSC als ABBs bezeichnet..

Das Ziel des *Real-Time Systems Compiler* ist es, aus dieser Beschreibung entweder ein ereignis- oder zeitgesteuertes System zu generieren. In beiden Fällen jedoch ist es notwendig, die maximale Ausführungszeit der einzelnen atomaren Einheiten, in diesem Falle also der ABBs, zu bestimmen.

Der in [18] vorgestellte *Real-Time Systems Compiler* nutzt dabei das LLVM-Framework und die LLVM-Zwischensprache (siehe Kapitel 2.2) als Basis, um ABBs darzustellen. Damit die Korrektheit des generierten Systems sichergestellt werden kann, müssen die maximalen Abarbeitungszeiten der einzelnen ABBs abgeschätzt werden. Für zeitgesteuerte Systeme wird diese Information benötigt, um die statische Ablaufplanung bei der Generierung des Systems zu ermöglichen. Bei ereignisgesteuerten Systemen wird die WCET benötigt, um die maximale Verzögerung bei der Antwort auf externe Ereignisse abschätzen zu können. Aus diesem Grund soll hier eine WCET-Analyse auf Ebene des LLVM-Zwischencodes realisiert werden, die mit vertretbarem Aufwand auf verschiedene, von der LLVM unterstützte Architekturen portierbar ist. Dabei sollen die konzeptionell höher liegenden Eigenschaften des Zeitverhaltens, wie z.B. die Anzahl Iterationen von Schleifen oder die Verzweigung des Kontrollflusses, mit den tieferliegenden zeitlichen Eigenschaften des generierten Assemblercodes in Verbindung gebracht werden. In diesem Zusammenhang ist es auch tolerierbar, dass durch den Ansatz auf einer etwas höheren konzeptionellen Ebene die Ergebnisse im Vergleich zu einer Analyse von reinem Objektcode etwas pessimistischer sind.

1.2 Aufbau der Arbeit

Diese Arbeit ist grob in drei große Teile gegliedert. Zuerst werden einige Grundlagen erläutert, die im Laufe der Arbeit verwendet werden. So wird ein Verfahren zur Bestimmung der WCET von bestimmten Programmrepräsentationen beschrieben. Daran schliesst sich eine Beschreibung des verwendeten LLVM-Frameworks und eine Analyse der Einschränkungen und Probleme, die bei einer WCET-Analyse im Bezug auf die LLVM zu beachten sind, an.

Im Anschluss daran folgt eine Beschreibung des Entwurfs und der Implementierung der WCET-Analyse. Nach einem groben Überblick über den Entwurf werden zunächst einige grundlegende Aspekte der Analyse beleuchtet und die damit einhergehenden Entwurfsentscheidungen erläutert. Schließlich wird die Entwurfsimplementierung beschrieben.

Daraufhin wird eine Evaluation des beschriebenen Ansatzes durchgeführt. Es werden anhand von verschiedenen Beispielanwendungen die mit der beschriebenen Implementierung erzielten Ergebnisse mit denen einer direkten Analyse von Objektcode verglichen.

Abgeschlossen wird die Arbeit durch eine kurze Ausführung über mögliche Verbesserungen und Erweiterungen des Verfahrens sowie die dazu notwendigen Schritte und Maßnahmen und durch eine kurze Vorstellungen von verschiedenen verwandeten Arbeiten zu diesem Thema.

2 Grundlagen und Voraussetzungen

In diesem Kapitel soll auf einige in dieser Arbeit verwendete grundlegende Algorithmen und Werkzeuge eingegangen werden. Dies ist zum einen ein Algorithmus zur Berechnung der maximalen Ausführungszeit von Programmen, zum anderen wird das verwendete LLVM-Framework vorgestellt sowie die für diese Arbeit relevanten Aspekte erläutert.

2.1 Einführung in die WCET-Analyse

Zu Beginn der Arbeit soll eine allgemeine Möglichkeit zur Bestimmung der WCET eines Programmstückes vorgestellt werden. Die WCET ist dabei die Zeit, die der Prozessor des betrachteten Systems maximal benötigt, um das Programmstück abzuarbeiten. Diese soll mit Hilfe einer statischen Analyse des ausgeführten Programmcodes bestimmt werden.

2.1.1 Statische WCET-Analyse

Mit Hilfe der statischen Analyse ist es möglich aus einer Beschreibung des Programmes und der Zielplattform dessen zeitliches Verhalten zu berechnen. Für die Programmbeschreibung existieren verschiedene Abstraktionsebenen. So ist es beispielsweise möglich einen Algorithmus in einer höheren Sprache wie C/C++ oder auch in Assemblersprache für einen beliebigen Prozessor zu beschreiben. Dabei ist die allgemeine Struktur der Programme immer ähnlich aufgebaut.

In Abbildung 2.1 auf der nächsten Seite ist ein einfaches C Programm abgebildet. Daneben ist der dazugehörige Kontrollflussgraph zu sehen. Dieser besteht aus einzelnen gerichteten Kanten und beschreibt, wie sich der Kontrollfluss bei der Abarbeitung durch das Programm bewegen kann. Die Knoten symbolisieren dabei die Basisblöcke, die wiederum die einzelnen Instruktionen enthalten. Basisblöcke haben die Eigenschaft, dass sämtliche in ihnen enthaltenen Instruktionen für jede Ausführung des Basisblocks genau einmal sequenziell abgearbeitet werden.

Für die WCET-Analyse bietet sich eine daran angelehnte Datenstruktur an: Der T-Graph (Timing Analysis Graph oder Zeitanalysegraph).

```
int gcd(int a, int b) {
    while (b != 0) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

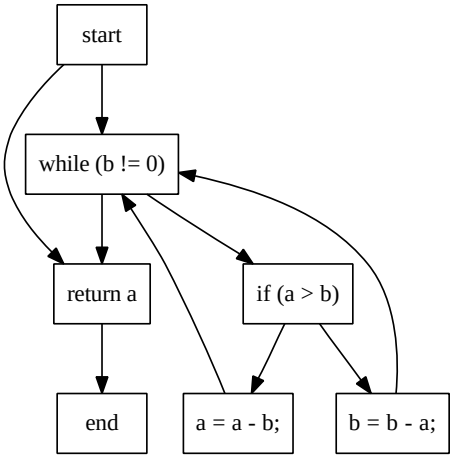


Abbildung 2.1: Einfache Funktion in C und der ihr entsprechende Kontrollflussgraph

2.1.2 Aufbau und Eigenschaften des T-Graphen

Ein T-Graph [17] ist ein gerichteter, zusammenhängender Graph G, der aus einer Menge Knoten $V = \{v_i : 0 \leq i < |V|\}$ und einer Menge gerichteter Kanten $E = \{e_i : 0 \leq i < |E|\}$ besteht. Er besitzt folgende Eigenschaften:

1. Es existiert genau ein Quellknoten, von dem nur Kanten wegführen.
2. Es existiert genau eine Senke, zu der nur Kanten hinführen.
3. Jede Kante ist Bestandteil von mindestens einem gerichteten Pfad, der von der Quelle zur Senke führt.
4. Es gibt eine Abbildung, die jeder Kante eine Zahl $\in \mathbf{N_0}$ zuordnet.

Jede Kante im T-Graph steht dabei für eine Abfolge von Instruktionen. Dabei ist nicht festgelegt, dass dies z.B. Assemblerinstruktionen sein müssen, sondern es lässt sich aus einer beliebig abstrakten Coderepräsentation ein T-Graph für die Zeitanalyse ableiten, solange man eine maximale Ausführungszeit für die einzelnen, sequenziellen Instruktionsfolgen angeben kann. Die Ausführung einer Kante entspricht dabei der Ausführung der ihr zugeordneten Instruktionen. Eigenschaften 1 und 2 stellen sicher, dass der T-Graph eine sinnvoll zu analysierende Einheit mit definiertem Prozeduranfang und -ende bildet. Eigenschaft 3 schließt Code, der vom Prozeduranfang aus nie erreicht werden kann, von der Analyse aus. Dies stellt sicher, dass nur korrekt Abarbeitungspfade betrachtet werden. Durch die Abbildung, die in Punkt 4 gefordert wird, wird jeder sequenziellen Instruktionsfolge ihre maximale Abarbeitungszeit zugewiesen.

Aus dieser Darstellung lassen sich nun die einzelnen Abarbeitungspfade des Programmes ableiten. Ein Abarbeitungspfad ist dabei ein Kantenzug, der von der Quelle zur Senke führt. Für

jeden Abarbeitungspfad P gibt es eine Ausführungszeit $xt(P)$, die sich über die Summe der Abarbeitungszeiten aller Kanten des Pfades berechnen lässt:

$$xt(P) = \sum_{i=0}^{|E|-1} f_{p_i} t_i$$

f_{p_i} steht dabei für die Anzahl der Ausführungen einer Kante bzw. der dazugehörigen Instruktionen, t_i für die Kosten einer einzelnen solchen Ausführung.

Damit lässt sich ein trivialer Algorithmus angeben, um die WCET eines Programmstückes zu berechnen: Durch Aufzählung aller möglichen Abarbeitungspfade des Programmes und der Berechnung der jeweiligen Abarbeitungszeit, lässt sich die WCET durch Bestimmen des Maximums berechnen. Diese Methode ist jedoch sehr ineffizient, da die Anzahl der möglichen Abarbeitungspfade mit steigender Programmgröße schnell zunimmt. Deshalb soll hier eine elegantere Lösung vorgestellt werden, die das WCET Problem auf die Bestimmung einer maximalen Zirkulation auf einem Graphen abbildet.

2.1.3 Zirkulationen auf T-Graphen

Eine Abbildung $f : E \to \mathbf{R}$ heißt Zirkulation auf dem Graphen G, wenn für jeden Knoten gilt, dass er genauso oft von eingehenden Kanten betreten wie er durch ausgehende Kanten verlassen wird. Der Fluss durch alle an einem Knoten eintreffenden Kanten ist also genauso groß wie der Fluss durch alle ausgehenden Kanten (Flusserhaltung):

$$\forall v \in V : \sum_{e:e^+=v} f(e) = \sum_{e:e^-=v} f(e)$$

e^+ steht dabei für den Anfang, e^- für das Ende einer gerichteten Kante. Es werden damit also die Kanten beschrieben, deren Fluss vom jeweiligen Knoten wegführt (e^+) bzw. hinführt (e^-).

Darüberhinaus benötigt man noch zwei weitere Abbildungen, die den Fluss durch eine Kante nach oben bzw nach unten beschränken: $b : E \to \mathbf{R}$ um nach unten, $c : E \to \mathbf{R}$ um nach oben hin zu begrenzen. Eine Zirkulation f heißt genau dann zulässig, wenn gilt:

$$\forall e \in E : b(e) \leq f(e) \leq c(e)$$

Zur Bestimmung der WCET eines durch einen T-Graphen beschriebenen Programmes mit Hilfe von Zirkulationen muss dieser in einen erweiterten T-Graphen transformiert werden. Dazu wird dem Graphen eine Rückkehrkante $e_{|E|}$ hinzugefügt, die von der Senke zur Quelle führt und welche die Abarbeitungszeit Null hat. Die Kapazitätsfunktionen werden wie folgt definiert:

$$b(e) = \begin{cases} 1 & e = e_{|E|} \\ 0 & \text{sonst} \end{cases}$$

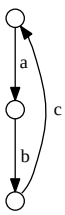
$$c(e) = \begin{cases} 1 & e = e_{|E|} \\ r(e) & \text{sonst} \end{cases}$$

Das Maximum der Abarbeitungen einer einzelnen Kante wird dabei als $r(e)$ bezeichnet. Die Kosten der Abarbeitung einer Kante entsprechen den Abarbeitungszeiten der jeweiligen Kante im T-Graph. Durch diese Transformation ergeben sich einige Konsequenzen für die Struktur und Interpretation des Zeitanalysegraphen:

Durch das Hinzufügen der Rückwärtskante wird der Graph geschlossen. Einer Abarbeitung des durch den Graphen repräsentierten Programmes entspricht nun ein geschlossener Kantenzug, der von der Quelle zur Senke führt und durch Inklusion der Rückwärtskante geschlossen wird. Der Abarbeitungspfad induziert hierbei einen ganzzahligen Fluss auf dem Graphen. Die Abarbeitung einer Kante inkrementiert hierbei den Fluss um 1. Dieser wird durch die beiden Funktionen $b(e)$ und $c(e)$ nach unten bzw. nach oben hin beschränkt. $b(e) \geq 0$ lässt den Fluss nicht negativ werden und $c(e) \geq 1$ sorgt dafür, dass keine Kante von vornherein den Fluss 0 annehmen kann. Für die Rückkehrkante gilt darüberhinaus $b(e_{|E|}) = c(e_{|E|}) = 1$. Durch die Flusserhaltung ist damit der Fluss von der Quelle weg bzw. zur Senke hin genau 1. Dadurch wird sichergestellt, dass eine Zirkulation genau einer Abarbeitung des Graphen entspricht.

Die nun folgenden Beispiele zeigen einige Ausschnitte aus erweiterten T-Graphen, die typisch für viele Kontrollstrukturen in Programmiersprachen sind.

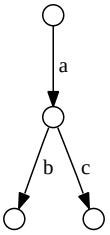
Sequenzen von Blöcken



Nebenstehend ist eine Sequenz bestehend aus den Kanten a und b inklusive der Rückkehrkante c zu sehen. Dies ist möglich, da in der Beschreibung des T-Graphen nicht gefordert wird, dass die Kanten Instruktionssequenzen maximaler Länge repräsentieren müssen. Durch die Flusserhaltung wird erzwungen, dass für jede Ausführung einer Kante ihre Nachfolgerkante abgearbeitet werden muss. Da die Rückkehrkante per Definition immer den Fluss 1 erzeugt, ist die Ausführungszeit für dieses Beispiel

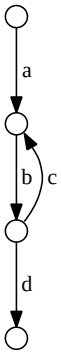
die Summe der Ausführungszeiten der drei Kanten, da durch die Flusserhaltung jede Kante genau einmal ausgeführt wird.

Verzweigung des Kontrollflusses



Hier ist eine Verzweigung zu sehen, wie sie typischerweise in vielen Programmiersprachen auftauchen kann. Die Flusserhaltung bedeutet hier, dass für jede Abarbeitung der Kante a nur entweder b oder c folgen können. Eine Zirkulation kann also eine Verzweigung beschreiben, indem sie für den einen Zweig einen Fluss größer Null und für den anderen Zweig einen Fluss gleich Null annimmt. Dabei muss immer $f(a) = f(b) + f(c)$ gelten.

Schleifen im Kontrollfluss



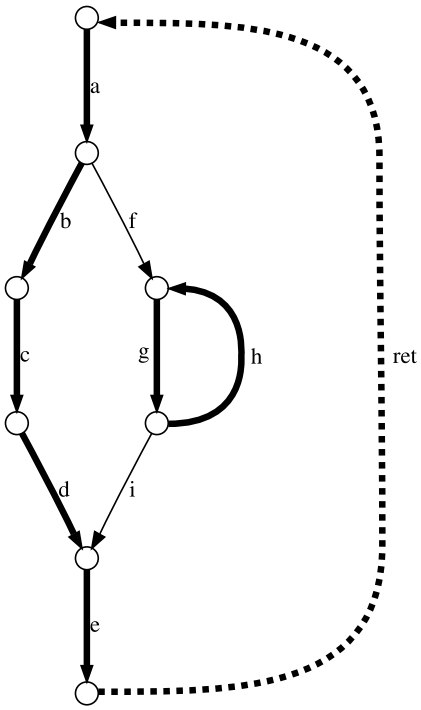
Der nebenstehende Ausschnitt aus einem T-Graph stellt mit einer Schleife eine weitere bedeutende Kontrollflusssequenz dar. Um eine maximale Abarbeitungszeit für dieses Beispiel angeben zu können, muss bekannt sein, wie oft die Schleife maximal durchlaufen werden kann. Im Allgemeinen ist es jedoch nicht möglich diese Information nur aus dem Programmcode zu extrahieren. In diesem Fall muss sie von außen vorgegeben werden, um eine WCET abzuschätzen. Diese Angabe bildet dann die maximale Kapazität der in der Schleife enthaltenen Kanten b und c. Läuft nun durch die Kante a ein Fluss $f(a) = 1$ und nimmt man an, dass die maximale Kapazität der Schleifenkanten b und c gleich $c(b) = c(c) = 10$ ist, so ergibt sich für den maximalen Fluss durch die Schleife folgendes:

Der Fluss durch Kante b nimmt den Wert 10 an, der durch Kante c lediglich den Wert 9. Dies wird dadurch bedingt, dass durch Kante d ebenfalls ein Fluss von 1 anliegen muss, da dort die Programmausführung fortgesetzt wird, die am Ende in die Rückkehrkante mit einem Fluss von 1 mündet.

2.1.4 Flussrestriktionen auf T-Graphen

Mit den bis hierhin vorgestellten Methoden kann man nun alle möglichen Abarbeitungen eines Programmes durch Zirkulationen auf einem erweiterten T-Graphen darstellen. Jedoch ergibt sich bei näherer Betrachtung ein Problem, insbesondere wenn der T-Graph nicht zyklenfrei

ist: Ein Zyklus alleine kann schon eine abgeschlossene Zirkulation bilden, auch wenn der in den Zyklus eingehende Fluss gleich Null ist. Dadurch stellt eine Zirkulation, die einen solchen Zyklus ohne eingehenden Fluss enthält, keine korrekte Abarbeitung dar. Die Zirkulation mit maximalen Kosten ist in einem solchen Fall lediglich eine grobe Abschätzung nach oben hin. Das Problem soll im Folgenden an einem Beispiel näher erläutert werden.



Kante e	$c(e)$	$\gamma(e)$	$f(e)$
a	1	10	1
b	1	20	1
c	1	15	1
d	1	25	1
e	1	10	1
f	1	12	0
g	5	10	5
h	5	10	5
i	1	14	0
ret	1	0	1

Abbildung 2.2: Problematische Zirkulation

In Abbildung 2.2 ist eine Zirkulation auf einem T-Graph zu sehen, auf die das eben beschriebene Problem zutrifft. Sie besteht aus dem Kantenzügen $(a,\ldots,e,ret)$ und (g,h), wobei der zweite einen Zyklus bildet. Diese in der Abbildung fett hervorgehobenen Kanten bilden den sog. Zirkulationssubgraphen. Er enthält die Kanten des erweiterten T-Graphen, für die $f(e) > 0$ gilt und alle Knoten, an denen mindestens eine dieser Kanten beginnt. Dieser Subgraph ist, wie man leicht erkennen kann, kein gültiger Abarbeitungspfad. Die Laufzeit dieser Zirkulation ist lediglich eine obere Schranke für die WCET des durch diesen T-Graphen beschriebenen Programmstückes. Diese Abschätzung ist jedoch nicht sehr genau, da sämtliche Zyklen im Graph zur Zirkulation maximaler Kosten hinzugezählt werden, unabhängig vom restlichen Graphen.

Die in diesem Beispiel dargestellte Zirkulation hat Kosten von 180:

$$\begin{aligned} Z &= \gamma(a)*f(a)+\gamma(b)*f(b)+\gamma(c)*f(c)+\gamma(d)*f(d)+\gamma(e)*f(e)+\gamma(f)*f(f)+ \\ &+ \gamma(g)*f(g)+\gamma(h)*f(h)+\gamma(i)*f(i)+\gamma(ret)*f(ret)=180 \end{aligned}$$

Wie man leicht feststellen kann ist der längste Abarbeitungspfad jedoch $(a,f,g,(h,g)^4,i,e,ret)$. Dieser hat lediglich Kosten von 136.

Graphentheoretisch heißt ein solcher Zirkulationssubgraph, wie er auf Abbildung 2.2 in fetten Kanten hervorgehoben ist, *nicht stark zusammenhängend*. Für die Bestimmung der WCET ist ein solcher Graph unvorteilhaft, da gültige Abarbeitungspfade immer stark zusammenhängend sein müssen. Eine gültige Zirkulation besitzt jedoch nicht automatisch diese Eigenschaft. Darum benötigt man noch zusätzliche Kriterien, um nur noch Zirkulationen zuzulassen, deren Zirkulationssubgraph stark zusammenhängend ist, und so einen gültigen Abarbeitungspfad für den T-Graphen beschreibt. Dadurch wird die Abschätzung der WCET weniger pessimistisch.

Dazu muss sichergestellt werden, dass Kanten in einem Zyklus des Graphen nur noch dann einen Fluss größer Null aufweisen dürfen, wenn Kanten mit einem Fluss größer Null existieren, die zum Zyklus hin- und wegführen. Dies lässt sich durch Ungleichungen bzw. Gleichungen beschreiben, die Flüsse durch verschiedene Kanten miteinander in Relation setzen:

$$\sum_{e_i \in E'} a_i f(e_i) \circ \sum_{e_i \in E'} a_i' f(e_i) + k$$

Solche (Un-)Gleichungen heißen auch Flussrestriktionen [17]. Dabei gilt $a_i, a_i' \in \mathbf{Z_0}$, $k \in \mathbf{N_0}$ und $\circ \in \{<, \leq, =\}$. Auch die bisher zur Beschränkung des Flusses verwendete Kapazitätsfunktion lässt sich durch eine geeignete Menge von Flussrestriktionen der Form $f(e) \leq c(e)$ bzw. $b(e) \leq f(e)$ ersetzen. Man benötigt jedoch nicht für jede Kante eine solche Restriktion, da sich durch die Flusserhaltung viele Kapazitätsangaben automatisch ergeben. So ist es beispielsweise nicht notwendig bei einer Sequenz von Kanten für jede eine Flussrestriktion aufzustellen, sondern es genügt bei einer einzigen. Durch die Flusserhaltung ist ein Fluss in den nachfolgenden automatisch beschränkt.

Für den in Abbildung 2.2 dargestellten T-Graphen lässt sich die Menge aller Zirkulationen, die gültigen Abarbeitungspfaden entsprechen, durch folgende Restriktionen darstellen.

$$f(ret) = 1$$
$$f(g) \leq 5f(f)$$

Die erste der beiden Restriktionen dient dazu den Fluss in der Rückkehrkante auf 1 festzulegen, so dass jede Zirkulation unter Einhaltung der Restriktionen genau einer möglichen Abarbeitung des beschriebenen Programmes entspricht. Die zweite sorgt nun dafür, dass der Fluss innerhalb einer Schleife mit dem in die Schleife eingehenden Fluss in Relation gesetzt wird. Damit wird das zu Beginn dieses Abschnittes beschriebene Problem gelöst und alle jetzt noch möglichen Zirkulationssubgraphen sind stark zusammenhängend und repräsentieren damit einen gültigen Abarbeitungspfad.

Somit ist es jetzt möglich nur unter Zuhilfenahme von Flussrestriktionen alle möglichen validen Abarbeitungspfade eines Programmes mit Hilfe von Zirkulationen auf einem erweiterten T-Graphen darzustellen. Nun wird noch ein möglichst effizientes Verfahren benötigt, um aus der Menge der möglichen Abarbeitungspfade den herauszusuchen, der die maximal mögliche Ausführungszeit hat. Dieses soll im folgenden aufgezeigt werden.

2.1.5 Bestimmung einer maximalen Zirkulation mit linearer Programmierung

Prinzipiell ist es, wie vorher schon beschrieben, möglich durch einfaches Aufzählen aller möglichen Abarbeitungspfade des Programmes und der Berechnung der jeweiligen dazugehörigen Ausführungszeit durch Maximumsbildung die WCET zu bestimmen. Aufgrund ihrer Ineffizienz ist diese Methode jedoch nicht sehr geeignet und skaliert insbesondere schlecht mit der Größe des Eingabeprogramms. Deshalb soll im Folgenden aufgezeigt werden, wie man die WCET mit Hilfe ganzzahliger linearer Programmierung bestimmt.

Lineare Programmierung

Ziel der linearen Programmierung ist es, den Wert einer Zielfunktion unter Einhaltung von verschiedenen Restriktionen zu maximieren [16]. Die Standardform eines linearen Programmierungsproblems sieht dabei folgendermaßen aus:

$$Z = c_1 x_1 + c_2 x_2 + \ldots + c_n x_n \tag{2.1}$$

Z ist eine lineare Funktion, deren Wert maximiert werden soll. Es gibt eine Menge von Restriktionen, um einen endlichen Maximalwert zu erhalten. Sie werden üblicherweise in Form eines linearen (Un-)gleichungssystems dargestellt:

$$a_{11}x_1 + a_{12}x_2 + \quad \ldots \quad + a_{1n}x_n \leq b_1 \tag{2.2}$$
$$a_{21}x_1 + a_{22}x_2 + \quad \ldots \quad + a_{2n}x_n \leq b_2 \tag{2.3}$$
$$\vdots$$
$$a_{m1}x_1 + a_{m2}x_2 + \quad \ldots \quad + a_{mn}x_n \leq b_m \tag{2.4}$$

mit

$$x_i \geq 0 \text{ für } 1 \leq i \leq m.$$

Die x_i heißen dabei die Entscheidungsvariablen des Modells. Das Ungleichungssystem stellt die Menge der Restriktionen dar. Ziel ist es, die Funktion unter Erfüllung all ihrer Restriktionen zu maximieren. Betrachtet man nur die ganzzahligen Lösungen der Zielfunktion, so spricht man von ganzzahliger linearer Programmierung.

Umwandlung des erweiterten T-Graphen in ein Programmierungsproblem

Die Umwandlung des erweiterten T-Graphen geschieht in mehreren Schritten:

- Als Zielfunktion dient die Kostenfunktion der Zirkulation:

  $$Z = \sum_{i=1}^{|E'|} f_i t_i$$

  Die t_i stehen dabei für die Abarbeitungszeiten der Kanten, die f_i beschreiben den Fluss durch die jeweiligen Kanten und bilden somit die Variablen des zum T-Graph gehörigen Programmierproblems.

- Die Menge von (Un-)Gleichungen, welche die Restriktionen bilden, ergibt sich aus dem Aufbau des T-Graphen und den zusätzlichen Einschränkungen für die Zirkulation. Um den Graph mit Hilfe von Restriktionen zu beschreiben, wird für jeden Knoten eine Gleichung der Form

  $$\sum_{e_j^+ = v_i} f_j = \sum_{e_k^- = v_i} f_k$$

  generiert.

- Die Restriktionen, die im Abschnitt 2.1.4 eingeführt wurden, um nur stark zusammenhängende Zirkulationen zuzulassen, können unverändert zur Menge der Restriktionen des Programmierungsproblems hinzugefügt werden.

- Für alle Variablen gilt dabei die *Nichtnegativitätsbedingung* $f_i > 0$. Sie wird in die Menge der Restriktionen mit aufgenommen.

Mit Hilfe dieser Richtlinien lassen sich beliebige T-Graphen in ganzzahlige lineare Programmierungsprobleme überführen und mit Hilfe von Standardsoftware zur Lösung von solchen Systemen lässt sich der maximale Fluss und damit die WCET des dazugehörigen Programmstückes berechnen. Aus der dabei erhaltenen Lösung geht nicht nur die maximale Abarbeitungszeit hervor, sondern aus der Belegung der Entscheidungsvariablen kann man auch noch ablesen wie oft die einzelnen Kanten im Ausführungspfad, der zur maximalen Abarbeitungszeit führt, abgearbeitet werden. Dadurch ist es möglich Programme zielgenau an den Stellen zu optimieren, an denen ein Großteil der Ausführungszeit verbraucht wird.

Lösen des Programmierungsproblems

Ein übliches Verfahren zur Lösung von linearen Programmierungsproblemen ist der *Simplex-Algorithmus*. Die Bibliothek *lp_solve* implementiert die davon abgeleitete Variante des *revidierten Simplexverfahrens*. Diese wird hier verwendet, um eine ganzzahlige Lösung der zu den T-Graphen äquivalenten Programmierungsprobleme zu finden. Für die Funktionsweise dieses Verfahrens sei auf [12, S.878] verwiesen.

Mit Hilfe der C-API kann man lp_solve die Gleichungen und Restriktionen in Form einer Matrix übergeben. Dabei kann man das (Un-)Gleichungssystem in verschiedenen Schritten aufbauen und so die Beschreibung des LP-Problems schrittweise vervollständigen. Eine genauere Beschreibung der API findet sich in [5].

2.2 Aufbau der LLVM

Im folgenden Kapitel soll die in dieser Arbeit verwendete *Low Level Virtual Machine* (LLVM) [4], in Zielsetzung und Entwurf erläutert und ihre Auswirkungen auf die WCET-Analyse aufgezeigt werden. Bei der Entwicklung der LLVM war es das Ziel eine einheitliche Plattform zur Übersetzung, Optimierung und Wartung von beliebiger Software über ihre komplette Lebenszeit hinweg zu schaffen. Folgende Kernmerkmale zeichnen LLVM aus, um dieses Ziel zu erreichen:

- Eine einheitliche, plattformunabhängige Zwischensprache mit einem sprachunabhängigen Typsystem

- Kapselung von sprach- und plattformspezifischen Eigenschaften wie Ausnahmebehandlung, Typkonversion und Adressarithmetik in einzelnen Instruktionen der Zwischensprache
- Verzicht auf ein bestimmtes Programmier- oder Speichermodell
- Keine Garantien zur Laufzeit bezüglich Typsicherheit und Speicherschutz

Die verwendete Zwischensprache ist von der Abstraktionsebene also nur wenig höher als ein Assembler für einen beliebigen Prozessor. Primär hat sie diesem lediglich die mitgeführten Typinformationen und eine Unterstützung von höherliegenden Konzepten wie z.B. der Ausnahmebehandlung mit Hilfe spezieller Instruktionen vorraus.

Die nun folgende Beschreibung gliedert sich in zwei großen Unterpunkten: Zuerst wird genauer auf die Zwischensprache des LLVM-Compilers eingegangen und daraufhin dann dessen Design genauer beschrieben.

2.2.1 LLVM-Zwischensprache

Die äußerste logische Einheit in LLVM bildet das Modul. Jedes Modul entspricht einer Übersetzungseinheit des Ursprungsprogrammes und besteht aus globalen Variablen, Funktionen und den sich daraus ergebenden Symbolen. Der eigentliche Programmcode befindet sich in den Funktionen des Moduls. Diese bestehen im Wesentlichen aus einer Ansammlung von Basisblöcken, deren Verzweigung untereinander den Kontrollflussgraph der Funktion bildet, und einer Beschreibung der Parameter und des Rückgabewerts. Die Basisblöcke wiederum sind aus einer Folge von einzelnen Instruktionen aufgebaut und werden von einer sog. Terminatorinstruktion abgeschlossen. Diese regelt die Verzweigung des Kontrollflusses zu weiteren Basisblöcken, wie im Fall von `br` oder `switch`, oder kennzeichnet das Ende der Funktion, wie im Fall von `ret`. Mit `invoke` und `unwind` ist es darüberhinaus möglich eine plattform- und quellsprachunabhängige Ausnahmebehandlung zu implementieren. Die `invoke`-Instruktion dient dabei dem Aufruf von Unterprogrammen, stellt dabei aber zwei Rückkehrpunkte zur Verfügung. Einer dient dabei der Fortführung des normalen Kontrollflusses, am anderen Punkt wird die Ausführung im Ausnahmefall fortgeführt: Kehrt die aufgerufene Funktion über eine `ret`-Anweisung zurück, so wird die Ausführung an der Sprungmarke für den normalen Kontrollfluss fortgeführt. Kehrt sie jedoch über ein `unwind` zurück, so springt die Ausführung zur Sprungmarke für den Ausnahmekontrollfluss, bei dem dann die Ausnahmebehandlung je nach Quellsprache unterschiedlich erfolgen kann. Für Subroutinenaufrufe ohne Ausnahmebehandlung dient die `call` Instruktion.

Der übrige LLVM-Instruktionssatz umfasst die wichtigsten Instruktionen eines gewöhnlichen Prozessors, abstrahiert jedoch von den maschinenspezifischen Eigenheiten. So wird statt einer

begrenzten Anzahl physikalischer Register eine unbegrenzte Anzahl von getypten, virtuellen Registern verwendet. Diese werden in *Static Single Assignment* Form (SSA Form) [9] beschrieben, d.h. jedes Register wird nur genau einmal geschrieben, kann aber beliebig oft gelesen werden. Der Speicher hingegen wird nicht in SSA Form angesprochen. LLVM ist eine typische Load-Store-Architektur, so dass nur die beiden Speicheroperationen `load` und `store` direkt auf den Speicher zugreifen können. Im Gegensatz zu vielen Assemblersprachen unterstützen die `load` und `store` Instruktionen jedoch keine indexierten Zugriffe auf den Speicher. Adressarithmetik wird explizit durch die `getelementptr` Instruktion derart durchgeführt, dass die Typinformationen dabei überprüft werden und intakt bleiben. Die dabei berechneten, getypten Zeiger dienen dann als Eingabe für die beiden Speicherzugriffsinstruktionen. Für die Allokation von Speicher existieren ebenfalls eigene Instruktionen: `malloc` und `free`. Sie agieren hierbei wie die aus C bekannten, gleichnamigen Bibliotheksfunktionen und dienen der Speicherverwaltung auf dem Heap. Zusätzlich gibt es in LLVM noch die Möglichkeit mit Hilfe von `alloca` direkt auf dem Stack ein Stück Speicher anzufordern. Dieses wird beim Verlassen der Funktion automatisch freigegeben. Für die binäre Arithmetik existieren die üblichen Instruktionen für Addition, Subtraktion, Multiplikation und Division. Darüberhinaus gibt es auch Befehle für bitweise, logische Operationen und Schiebeoperationen. Für den Vergleich von zwei Variablen existiert jeweils für ganzzahlige und Gleitkommavariablen eine Instruktion.

Eine Besonderheit des LLVM-Instruktionssatzes im Vergleich zu üblichen Assemblerbefehlssätzen für Prozessoren ist die sog. `PHI` Instruktion. Die Register werden, wie vorher bereits erwähnt, in SSA Form verwaltet. Daher ist es notwendig bei der Konvertierung von Programmcode in die SSA Form mehrere schreibende Zugriffe auf dieselbe Variable auf mehrere Variablenkopien aufzuteilen. Die `PHI` Instruktion dient nun dazu, beim Eintritt des Kontrollflusses in einen neuen Basisblock in Abhängigkeit vom vorher ausgeführten Block eine bestimmte Kopie einer Variable auszuwählen.

In Abbildung 2.3 ist eine einfache Schleife in LLVM zu sehen. Der Laufparameter, der bei jedem Durchlauf der Schleife inkrementiert wird, ist `%indvar`. Die SSA-Form verhindert darauf jedoch schreibende Zugriffe. Deshalb wird eine Kopie, `%indvar.next`, erzeugt, die den um eins inkrementierten Laufparameter enthält. Am Ende des Basisblockes `bb` wird wieder zu `bb` zurückgesprungen oder, wenn die Abbruchbedingung erfüllt ist, die Schleife verlassen. Die `PHI`-Instruktion am Beginn des Basisblocks dient nun dazu die richtige Version des Laufparameters auszuwählen. Wird die Schleife zum ersten Mal betreten, so wird er mit Null initialisiert, ansonsten nimmt `%indvar` den Wert von `%indvar.next` an. Auf diese Weise kann man die in vielen Programmiersprachen übliche Semantik der beliebig schreib- und lesbaren Variablen nachbilden.

Zusätzlich zu den festen LLVM-Befehlen existiert noch eine Vielzahl von *Intrinsics*. Diese

```
bb.pre:
  ...
  br label %bb

bb:
  %indvar = phi i32 [ 0, %bb.pre ], [ %indvar.next, %bb ]
  %indvar.next = add i32 %indvar, 1
  %exitcond = icmp eq i32 %indvar.next, 10
  br i1 %exitcond, label %bb8, label %bb

bb8:
  ...
```

Abbildung 2.3: Einfache Schleife mit `PHI`-Instruktion

können wie normale Funktionen aufgerufen werden, aber sind dabei nach einem festen Namensschema benannt. Sie dienen der leichten Erweiterbarkeit des Befehlssatzes, da durch ihre Implementierung als Funktionen beim Hinzufügen weiterer *Intrinsics* der Codegenerator nicht angepasst werden muss. Die den Intrinsics entsprechenden Funktionen liegen in LLVM-Code vor und werden vom Codegenerator wie normale Funktion mit übersetzt.

2.2.2 LLVM-Infrastruktur

Die Übersetzung eines Programmes mit Hilfe des LLVM-Frameworks läuft grob in drei Schritten ab: Zuerst wird der zu übersetzende Programmcode von einem sprachspezifischen Frontend in die LLVM Repräsentation überführt. Nachdem das Programm nun in einer von der Eingabesprache unabhängigen Zwischensprache vorliegt, kommen alle Optimierungen, die darauf ausgeführt werden, allen Frontends zugute. Nach dieser Optimierungsphase wird der LLVM-Code von einem plattformspezifischen Backend in die Zielsprache(z.B. Assembler für x86 oder PowerPC) übersetzt. Im Folgenden sollen diese drei Phasen der Codegenerierung näher erläutert werden.

Frontend

Ein Frontend ist dafür verantwortlich Konstrukte der Eingabesprache in LLVM-Zwischencode zu übertragen. Darüberhinaus müssen Optimierungen, die sich explizit auf die Eingabesprache beziehen, ebenfalls im Frontend ausgeführt werden. Dabei können auf die Quellsprache

zugeschnittene Algorithmen und Datenstrukturen verwendet werden. LLVM benutzt zum jetzigen Zeitpunkt (07-2008) ein adaptiertes GCC Frontend, welches modifiziert wurde, um LLVM-Zwischencode zu erzeugen, da manche Sprachen wie z.B. C++ sehr schwer zu parsen sind. Für die weitere Arbeit ist das Frontend von relativ geringer Bedeutung, weil sich die Analyse auf die LLVM-Zwischensprache bezieht.

Optimierung auf LLVM-Zwischencode Ebene

Einen wichtigen Teil des LLVM-Frameworks stellt die plattformunabhängige Optimierungsarchitektur dar. Sie operiert direkt auf LLVM-Zwischencode. Die einzelnen Optimierungsschritte sind dabei in sog. *Passes* aufgeteilt, deren Ausführung prinzipiell in beliebiger Reihenfolge geschehen kann.

Es gibt dabei im Wesentlichen zwei verschiedene Arten von Passes: Wird keine Veränderung am Zwischencode vorgenommen, sondern lediglich Informationen aus der LLVM-Zwischensprache gewonnen, so spricht man von einer Analyse. Ein Beispiel dafür ist die Konstruktion von natürlichen Schleifen aus dem LLVM-Zwischencode. Die zweite Klasse von Passes stellen die Optimierungen dar. Sie verändern den LLVM-Zwischencode nach bestimmten Regeln, um ihn auf ein bestimmtes Ziel, wie die Laufzeit oder die Codegröße, hin zu optimieren. Dazu zählen Techniken wie die Optimierung von Schleifen durch Verschieben von invariantem Code aus dem Schleifenrumpf heraus, oder die Elimination von totem Programmcode. Für die Berechnung ihrer Ergebnisse können Passes auf die Ergebnisse von vorhergehenden Analysen zurückgreifen. Beispielsweise benötigen viele Algorithmen zur Schleifenoptimierung Informationen über die im analysierten Programm vorliegenden Schleifenkonstrukte. Die Analyse, die diese Informationen aus dem Programm erzeugt, muss also vor der Optimierung ausgeführt werden, die sie dann verarbeitet. Desweiteren kann es passieren, dass die zuvor berechneten Schleifeninformationen dann nach der Optimierung nicht mehr mit dem nun veränderten Programm übereinstimmen. Sollte also ein weiterer Pass diese Informationen benötigen, so müssen sie neu berechnet werden.

Diese Aufgabe übernehmen innerhalb des LLVM-Frameworks die *PassManager*. An diesen Objekten im LLVM-Framework kann man die einzelnen Passes registrieren. Die von einem Pass gemeldeten Abhängigkeiten werden dann vom Passmanager berücksichtigt und die Reihenfolge der Abarbeitung dann so geplant, dass die Passes auf die Ergebnisse ihrer Vorgänger zugreifen können. Sollten Ergebnisse von Analysen durch die Ausführung von Optimierungspasses invalidiert werden, so werden sie bei Bedarf neu berechnet, um weiteren Optimierungen zur Verfügung zu stehen.

Dazu orthogonal existiert noch eine weitere Aufteilung der Passes in Kategorien, die festlegen, inwieweit ein Pass ein Programm modifizieren darf. Die allgemeinste Form ist der *ModulePass*. Dieser läuft über ein komplettes Übersetzungsmodul und kann dort Funktionen oder globale Variablen hinzufügen, entfernen oder modifizieren. Eine Stufe spezifischer ist der *FunctionPass*, der lediglich einzelne Funktionen des Programmes modifizieren kann. Schließlich gibt es auch noch einen *BasicBlockPass*, der Änderungen nur innerhalb eines einzelnen Basisblocks vornehmen kann. Unabhängig von dieser Hierarchie existiert noch der *CallGraphSCCPass*, der dazu dient den Aufrufgraphen des Programmes von unten nach oben zu traversieren und der *LoopPass*, mit dem es möglich ist über alle Schleifen eines Programmes zu iterieren.

Im Rahmen des LLVM-Frameworks gibt es eine vielzahl solcher Optimierungsalgorithmen in Form von Passes, die je nach Anforderung der Anwendung auf dem Eingabeprogramm ausgeführt werden können. Dies sind alles LLVM-zu-LLVM-Transformationen, und damit sowohl quell- als auch zielsprachunabhängig.

Übersetzen des LLVM-Zwischencodes in plattformspezifischen Assembler

Der auf LLVM-Ebene optimierte Code muss nun noch auf die Zielarchitektur übersetzt werden, um ihn dort ausführen zu können. Diese Aufgabe übernehmen innerhalb des LLVM-Frameworks die *Backends*. Sie sind zwar für jede Zielarchitektur unterschiedlich, jedoch ist der allgemeine Aufbau meist ähnlich. Die Arbeit der Backends ist analog zu den vorher besprochenen Optimierungen in einzelne Passes aufgeteilt. Dies sind die Kernaufgaben des Backends:

- Auswahl von prozessorspezifischen Instruktionen oder Instruktionssequenzen für die einzelnen LLVM-Befehle.
- Abbildung der unbegrenzten Anzahl von virtuellen Registern in LLVM auf eine begrenzte Anzahl von Registern des Zielprozessors.
- Erzeugung von Assembler- oder Objektcode.

Zusätzlich dazu können im Backend noch plattformspezifische Optimierungen ausgeführt werden, um den Zielprozessor möglichst gut auszunutzen. Dazu zählt z.B. die Ablaufplanung für einzelne Instruktionen, um auf die Eigenschaften der Zielarchitektur bezüglich Anzahl und Anordnung der Ausführungseinheiten oder Pipelineeffekte zu optimieren. Für die Beschreibung der Ausgabesprache, typischerweise Assembler für einen Zielprozessor, muss hier jedoch eine für das jeweilige Ziel angepasste Repräsentation verwendet werden. Um diese aufwendige, und für jedes Backend zu wiederholende Arbeit zu erleichtern, verwendet LLVM das Tool *TableGen*. Vereinfacht gesagt, nimmt es eine Beschreibung der Assemblersprache des Zielprozessors

entgegen und benutzt diese, um die Klassenstruktur für die jeweilige Coderepräsentation zu erzeugen. Ebenfalls auf diesem Weg wird ein Instruktionsselektor erzeugt.

Ab dem Zeitpunkt der Instruktionsauswahl existiert das Programm in zwei Formen: Zum einen liegt es noch im plattformunabhängigen Zwischencode vor, zum anderen wird bei der Instruktionsauswahl ein maschinenspezifisches Instruktionsformat erzeugt. Für die Ausführung von Passes auf dieser Maschinencoderepräsentation existiert der *MachineFunctionPass*. Alle Operationen, die nach dem Ausführen der Instruktionsauswahl geschehen sollen, sind eine Unterklasse eines solchen und operieren auf den erzeugten Maschineninstruktionen. Auf diese Weise kann man weitere plattformspezifische Optimierungen im Backend ausführen. Am Ende des Codeerzeugungsprozesses steht dann entweder ein Assemblerdrucker, der menschenlesbaren Assemblercode erzeugt oder ein Pass, der aus der Maschinencodebeschreibung direkt gebundenen und auf der Maschine ausführbaren Objektcode erzeugt.

2.3 Analyse der LLVM bezüglich WCET Berechnung

Im Folgenden soll das komplette LLVM-Framework bezüglich der WCET-Analyse auf Ebene des LLVM-Zwischencodes betrachtet werden. Dies ist aufgeteilt in eine Betrachtung der LLVM-Zwischensprache und des dazugehörigen Compilerframeworks.

2.3.1 LLVM-Zwischensprache

Möchte man die WCET eines Programmes auf Ebene der LLVM-Zwischensprache mit Hilfe des in Kapitel 2.1 vorgestellten Verfahrens bestimmen, so nimmt man den Kontrollflussgraph, der durch die Anordnung der LLVM-Basisblöcke gegeben ist, und transformiert ihn in einen erweiterten T-Graphen. Dabei muss man jedoch davon ausgehen, dass sich bei der weiteren Übersetzung des LLVM-Zwischencodes in Maschinencode der Kontrollflussgraph nicht mehr verändert. Insbesondere die einzelnen Instruktionen dürfen sich bei der Transformierung von LLVM-Zwischencode zu nativem Assemblercode nicht in komplexere Instruktionssequenzen mit Schleifen verwandeln. Dies ist jedoch bei einigen Zielarchitekturen nicht unbedingt möglich. So bietet die LLVM-Zwischensprache beispielsweise direkte Befehle für Multiplikation und Division, die vor allem auf kleineren Mikrocontrollern nicht unbedingt in Hardware implementiert sind und so durch eine Folge von einfacheren Assemblerinstruktionen ersetzt werden müssen. Sind diese Operationen als Schleifen implementiert, so ist es nicht mehr einfach möglich deren maximale Ausführungszeit zu bestimmen.

Theoretisch ist es denkbar, dass jede LLVM-Instruktion auf eine solche Weise durch ein Backend implementiert wird, da LLVM selbst die Art der Implementierung nicht vorschreibt. Dies betrifft vor allem das zeitliche Verhalten der einzelnen Instruktionen. In der Realität wird aber meist auf eine möglichst effiziente Implementierung wert gelegt. Man muss jedoch das Backend der LLVM, welches man für die WCET-Analyse verwenden möchte, auf solche Instruktionen hin untersuchen, um solche Probleme zu vermeiden.

Potentiell dafür in Frage kommende Instruktionen sind komplexere arithmetische Befehle wie Multiplikation und Division, die, wie schon vorher erwähnt, nicht immer direkt auf einfache Assemblerbefehle abgebildet werden können. Je nach Ausrichtung und Komplexität der Zielarchitektur und des Zielprozessors ist dies bei verschieden vielen arithmetischen Instruktionen der Fall. Darüberhinaus sind insbesondere die Instruktionen zur Speicherverwaltung problematisch. Bei `malloc` und `free` hängt die Abschätzbarkeit der Ausführungszeit entscheidend von den zur Implementierung benutzten Datenstrukturen ab. Auch für `alloca` ist es vorstellbar, dass eine komplexere Speicherverwaltungsmethode implementiert wird, deren Ausführungszeit nicht einfach nach oben abgeschätzt werden kann. Die verwendeten Speicherverwaltungsalgorithmen sind dabei spezifisch für das jeweils verwendete Frontend, welches die semantischen Vorgaben zur Speicherverwaltung, die durch die Quellsprache vorgegeben sind, mit Hilfe der LLVM umsetzt.

Zwei weitere Instruktionen, deren Ausführungszeit nicht auf einfache Weise bestimmt werden kann, sind `call` und `invoke`. Ihre Ausführungszeit hängt entscheidend davon ab wie lange die aufgerufene Funktion zur Ausführung benötigt. Diese muss also ebenfalls analysiert werden und ihre WCET wird dann zur Ausführungszeit der aufrufenden Instruktion addiert. Im Falle von `invoke` kommt auch noch hinzu, dass sich der Kontrollfluss hier potentiell verzweigen kann und man bei der WCET-Analyse auch immer noch den Ausnahmefall mit berücksichtigen muss. Ein weiteres Problem ergibt sich in diesem Kontext, sobald die aufgerufene Funktion rekursiv wiederum sich selbst aufruft oder sogar Teil eines komplexeren Aufrufzykluses ist. Die Rekursionstiefe hängt oft von den Eingabeparametern der jeweiligen Funktion ab und kann so meistens nicht zur Übersetzungszeit bestimmt werden. In diesem Fall muss die Rekursionstiefe von außen vorgegeben werden, um eine WCET-Analyse zu ermöglichen. Die übrigen LLVM-Zwischencodeinstruktionen sind dagegen meist unproblematisch und werden in einfachen, sequenziellen Code der Zielarchitektur übersetzt. Somit stehen sie einer WCET-Analyse nicht im Wege.

Es ist also notwendig im Vorfeld einer möglichen WCET-Analyse alle problematischen LLVM-Instruktionen für das jeweils verwendete Backend zu identifizieren und dann nach einer Abschätzung für ihre Ausführungszeit zu suchen. Gelingt dies, so ist es möglich die Berechnung der WCET auf der Ebene des LLVM-Zwischensprache durchzuführen.

2.3.2 LLVM-Compilerframework

Wie in Abschnitt 2.2.2 beschrieben sind die verschiedenen Arbeitsschritte, die im Laufe der Programmübersetzung ausgeführt werden, in einzelne Passes aufgeteilt. Nachdem das Programm in die LLVM-Zwischensprache überführt wurde, findet auf dieser der Großteil der Optimierungen statt. Dabei stellt sich die Frage, an welcher Stelle des Optimierungsprozesses man den Zwischencode bezüglich der WCET betrachten möchte, denn ab dieser Stelle muss der Kontrollflussgraph so weit wie möglich invariant bleiben. Er darf also durch die Passes, die bis zum Ende der Codegenerierung ausgeführt werden, nicht mehr verändert werden. Dies schränkt die Auswahl an Optimierungsmöglichkeiten weitestgehend ein. So ist dann beispielsweise *Loop Unrolling* nicht mehr möglich, da hier der Kontrollflussgraph des Programmes verändert wird.

Die Berechnung der WCET kann man dabei gut in einzelne Passes aufteilen, die sich dann nahtlos in das Passsystem von LLVM integrieren lassen. So muss aus dem LLVM-Zwischencode ein erweiterter T-Graph erzeugt werden und, nachdem dann vom Backend für den jeweiligen Basisblock Assemblercode produziert wurde, die Ausführungszeit des Basisblocks bestimmt und als Kantengewicht im T-Graph festgehalten werden. Die T-Graph-Erzeugung ist dabei noch von der jeweiligen Zielplattform unabhängig, da der T-Graph direkt aus dem LLVM-Zwischencode generiert wird. Die Bestimmung der Ausführungszeit für die einzelnen Basisblöcke erfordert jedoch eine Änderung des Backends für die jeweilige Plattform. Dieses muss derart modifiziert werden, dass die Codeerzeugung nicht mehr für komplette Funktionen ausgeführt wird, sondern für jeden Basisblock einzeln Assemblercode generiert wird.

2.4 aiT-Analyzer

Das aiT-WCET-Analysewerkzeug [1] ist ein Programm zu Abschätzung der WCET von Objektcode für verschiedene Prozessormodelle, die in eingebetteten Systemen Verwendung finden. In dieser Arbeit kommt die Version für einen PowerPC MPC565 [6] zur Anwendung. Das Ziel von aiT ist es dabei die WCET für das Zielsystem so wenig wie möglich zu überschätzen. Dies ist relativ schwer zu erreichen, da moderne Prozessoren typischerweise Pipelines und Caches verwenden, um eine möglichst hohe Performanz zu erzielen. Deswegen greift das aiT-WCET-Analysewerkzeug zu einer Reihe von Techniken, um die Ausführung der einzelnen Instruktionen auf der Hardware so realistisch und genau wie möglich zu simulieren. So werden die Effekte, die Pipelines und Caches auf die Ausführung von Programmcode haben, bei der WCET-Analyse mit berücksichtigt. Dadurch ist die Qualität der vom aiT durchgeführten Analysen sehr gut und im Allgemeinen sehr nah an den realen maximalen Ausführungszeiten der vermessenen Programme.

In dieser Arbeit findet er als Werkzeug zur Bestimmung von Basisblockausführungszeiten und als Vergleichsmaßstab für die mit der hier vorgestellten Methodik erreichten WCET Messwerte.

2.5 Zusammenfassung

In diesem Kapitel wird zuerst eine Methode vorgestellt, um für eine Programmbeschreibung die maximale Ausführungszeit zu bestimmen. Zu diesem Zweck wird der T-Graph als Datenstruktur vorgestellt, der die für die Zeitanalyse relevanten Informationen des Programmes kapselt. Durch Konvertierung in einen T-Graphen ist es möglich, für eine Vielzahl von Programmbeschreibungen eine Zeitanalyse durchzuführen. Auf dem T-Graphen werden dann die möglichen Abarbeitungen eines Programmes als Zirkulation beschrieben und durch Bestimmung der Zirkulation mit maximalen Kosten die WCET des dazugehörigen Programmstückes ermittelt. Dies geschieht durch die Konvertierung des T-Graphen in ein lineares Programm und die Lösung dessen mit Hilfe darauf spezialisierter Softwarepakete.

Danach folgt eine Beschreibung der als Basis verwendeten LLVM. Dabei wird zuerst der Aufbau der verwendeten LLVM-Zwischensprache und anschließend der Aufbau des dazugehörigen Frameworks erläutert. Schließlich folgt noch eine Betrachtung der LLVM im Kontext der WCET-Analyse. Dabei wird kurz auf mögliche Probleme, die durch die LLVM verursacht werden, eingegangen.

3 Design und Implementierung

Im nun folgenden Kapitel soll das Design der WCET-Analyse für das LLVM-Framework erläutert und daraufhin Details der konkreten Implementierung beschrieben werden. Zuerst werden dabei einige grundlegende Überlegungen aufgezeigt, die den Entwurf der Implementierung maßgeblich beeinflussen.

3.1 Grundlegende Designüberlegungen

3.1.1 Ansatz zur WCET-Analyse

Die Analyse der WCET kann auf unterschiedlichen Ebenen des Codeerzeugungsprozesses stattfinden: Zum einen kann man den generierten Assembler- oder Objektcode betrachten. Dazu ist es notwendig die Struktur des Programmes aus den einzelnen Assemblerinstruktionen zu rekonstruieren. Danach ist es dann möglich mit Hilfe der in Kapitel 2.1 vorgestellten Methodik und einer Möglichkeit, die Basisblockausführungszeiten zu bestimmen, die WCET zu berechnen. Diese Vorgehensweise hat den Vorteil, dass man dabei von der ursprünglichen Repräsentation des Programmes und des Compilers, mit dem diese in Assemblercode übersetzt wird, unabhängig ist. Dagegen ist die Analyse in ihrer Ganzheit von der jeweiligen Zielarchitektur abhängig.

Will man eine portablere Analyse durchführen, so bietet es sich an, diese in den Übersetzungsprozess mit einzubinden und die Analyse auf der internen, plattformunabhängigen Repräsentation des Übersetzers durchzuführen. Dabei verliert man allerdings die Unabhängigkeit vom Compiler. Dies ist jedoch nicht so schwerwiegend, da das in dieser Arbeit betrachtete LLVM-Framework Assembler- und Objektcode für eine Vielzahl von verschiedenen Prozessorarchitekturen generieren kann. Dies gilt auch für viele andere Compiler wie z.B. die GCC. Man gewinnt also durch eine Analyse, die in den Übersetzungsprozess eingebunden ist, eine leichte Portierbarkeit auf verschiedene Zielarchitekturen, die jedoch alle vom verwendeten Compilerframework unterstützt werden müssen.

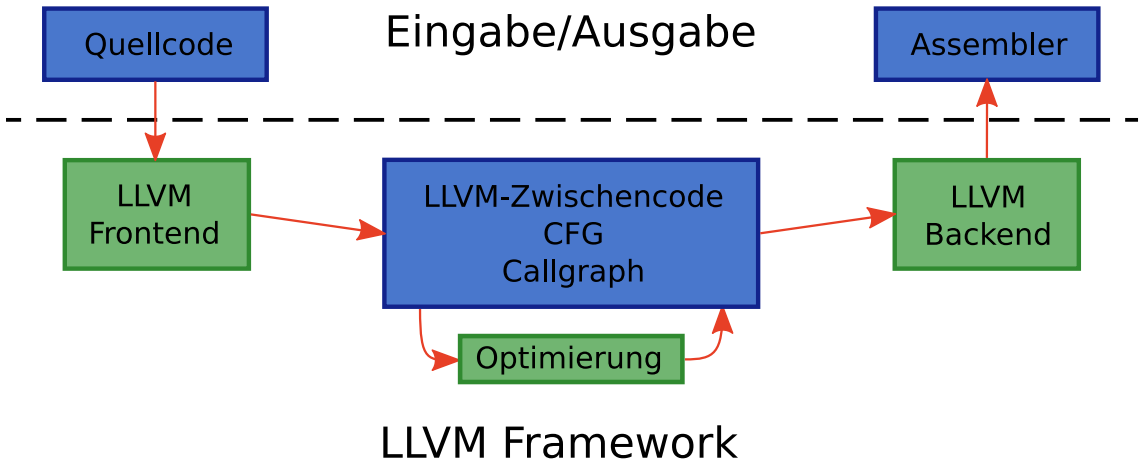


Abbildung 3.1: Grobe Struktur des LLVM-Frameworks

In 3.1 ist die grobe Struktur des verwendeten LLVM-Frameworks zu sehen. Blau hinterlegt sind die Daten/Datenstrukturen, die Algorithmen, die darauf ausgeführt werden, sind grün hervorgehoben. Der Quellcode des Eingabeprogrammes wird vom LLVM-Frontend in die LLVM-Zwischensprache (siehe Kapitel 2.2.1) umgewandelt. Auf dieser können dann Optimierungen ausgeführt werden und am Ende dieses Prozesses wird die LLVM-Zwischensprache in den Assembler der Zielplattform konvertiert. Die Analyse soll nun auf der internen Programmrepräsentation des LLVM-Frameworks durchgeführt werden, die den Kontrollfluss durch die Basisblöcke und die einzelnen Operationen innerhalb dieser beschreibt. Die Analyse setzt an dieser Stelle an, weil dort noch sämtliche Informationen aus dem Eingabeprogramm vorhanden sind. Aus dem LLVM-Zwischencode werden dabei im Laufe der Analyse sämtliche für die Zeitanalyse notwendigen Informationen extrahiert und dann mit deren Hilfe die WCET des betrachteten Codestückes berechnet.

Aufbauend auf dem Design der LLVM ist in Abbildung 3.2 der Entwurf der WCET Analyse zu sehen. Zentraler Bestandteil dieser ist der T-Graph, der aus dem LLVM-Zwischencode erzeugt wird. Zu Beginn bildet dieser lediglich den Kontrollflussgraphen aus der LLVM-Zwischensprache ab. Im Lauf der Analyse jedoch wird er mit allen zur Zeitanalyse notwendigen Informationen angereichert. Dazu gehören Schranken für die Anzahl von Schleifenausführungen, die entweder automatisch aus dem Zwischencode generiert werden können oder manuell mit Hilfe von Annotationen bereitgestellt werden. Auch muss für jeden Unterprogrammaufruf festgestellt werden, ob er rekursiv ist und dies der Analyse bekannt gemacht werden.

Durch die Codegenerierung im Backend des LLVM-Frameworks erhält man schließlich eine Assemblerrepräsentation für jeden einzelnen Basisblock der Zwischensprache. Diese kann nun

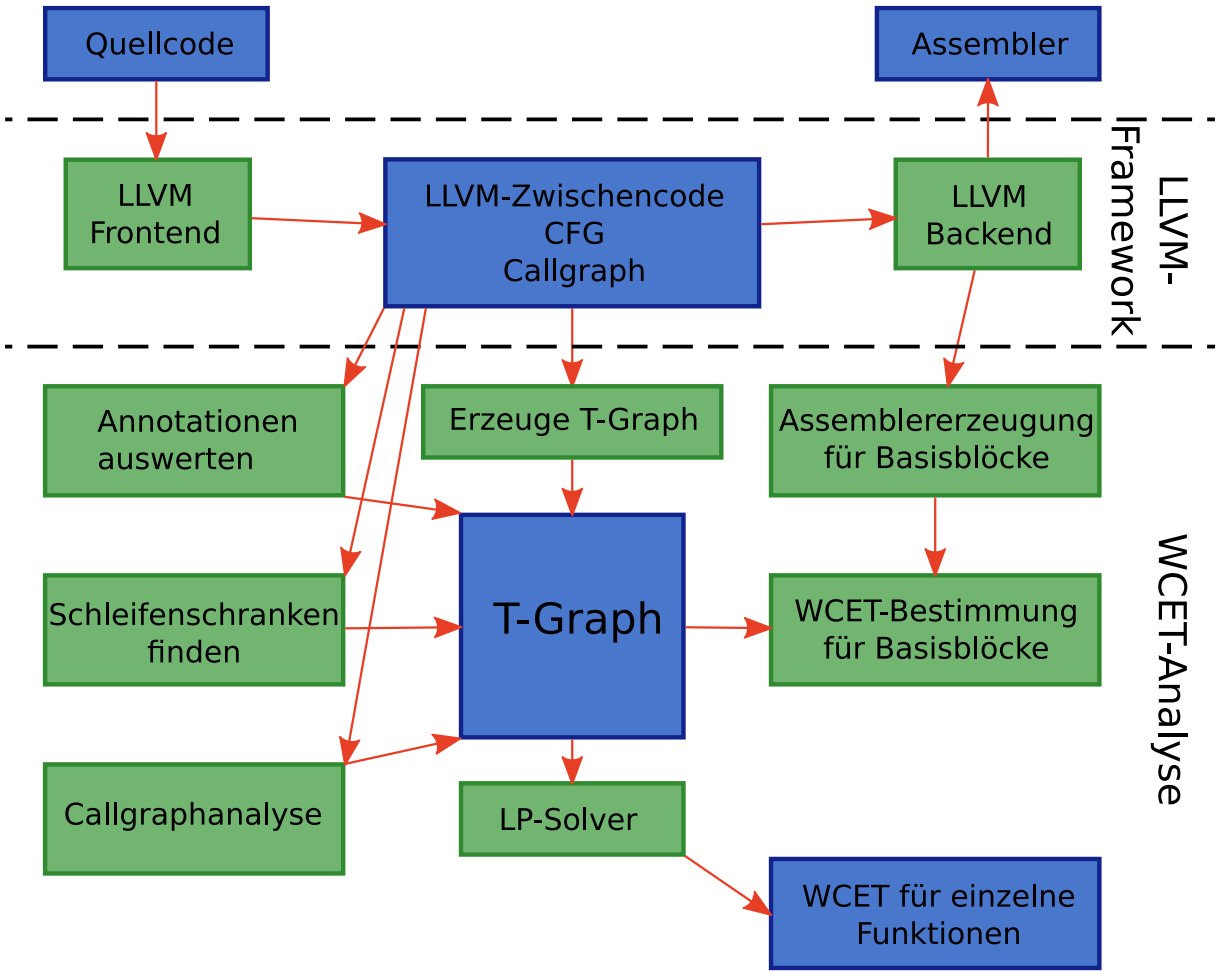


Abbildung 3.2: Entwurfsüberblick

durch ein externes Programm auf seine Ausführungszeit hin analysiert werden. Das ist relativ einfach möglich, da sich in den Basisblöcken keine komplexeren Strukturen wie beispielsweise Schleifen befinden, und somit nur eine einfache Sequenz von Instruktionen analysiert werden muss. Durch die Verwendung eines externen Programmes spart man sich jedoch ein eigenes, je nach verwendeter Hardware sehr komplexes Zeitmodell für die einzelnen Instruktionen des Zielprozessors generieren zu müssen. Mit den nun bekannten Basisblockausführungszeiten ist es möglich mit Hilfe des in 2.1 vorgestellten Verfahrens die WCET für eine derart übersetzte Funktion zu berechnen.

In den folgenden Kapiteln sollen nun einige der hier dargestellten Problemstellungen näher untersucht und verschiedene Lösungsmöglichkeiten dargelegt werden.

3.1.2 Ausführungszeiten von Basisblöcken

Ein Kernstück der Analyse auf Ebene des LLVM-Zwischencodes ist die Art und Weise, wie man für die einzelnen Basisblöcke bestimmt, welche maximale Ausführungszeit sie auf der gewählten Zielhardware haben werden. Für die Analyse von Assemblercode der Zielplattform steht mit dem aiT-Analyzer (siehe Kapitel 2.4) ein externes Werkzeug zur Verfügung, das für die Bestimmung der WCET der Basisblöcke verwendet werden soll. Ein Ansatz zur Generierung des Assemblercodes für diese Analyse ist es, die einzelnen Basisblöcke getrennt voneinander zu übersetzen. Es ist allerdings innerhalb des LLVM-Frameworks nicht möglich, einzelne Basisblöcke durch den Übersetzungsprozess zu schicken. Dieser kann nur von kompletten Funktionen durchlaufen werden. Deshalb ist es notwendig, aus den einzelnen Basisblöcken Funktionen zu generieren. Dabei muss jedoch beachtet werden, dass ein Basisblock alleine nicht automatisch eine legale LLVM-Funktion bildet. Die Basisblöcke eines Programmes greifen typischerweise auf eine gemeinsame Menge von Variablen zu. Es kann also vorkommen, dass eine Instruktion auf eine Variable, die in einem anderen Basisblock definiert wurde, lesend zugreift. Schreibende Zugriffe sind aufgrund der in SSA-Form verwalteten Register nicht möglich. Trennt man den Verbund der Basisblöcke nun auf und packt jeden davon in eine eigene Funktion, so muss man dafür sorgen, dass verwendete Variablen aus anderen Basisblöcken durch eine *Alibiversion* ersetzt werden, um korrekten LLVM-Code zu erhalten. Übersetzt man nun diese Funktion mit den Werkzeugen des LLVM-Frameworks, dann resultiert daraus ein schleifenfreies Assemblercompilat, das man ohne große Probleme vom aiT-Analyzer untersuchen lassen kann, um die WCET zu bestimmen.

In Abbildung 3.3 ist ein Beispiel für einen einzelnen Basisblock in LLVM zu sehen. Problematische Stellen sind die `PHI`-Instruktion am Anfang in Zeile 1 und die Verwendung der Variablen `a` und `b` in den Zeilen 4 und 5. Die Änderungen, die dazu nötig sind, um daraus eine gültige LLVM-Funktion zu generieren, dürfen sich auf die Ausführungszeit des Basisblocks nicht verbessernd auswirken. Es ist also notwendig jeweils die Aktion zu wählen, welche die schlimmstmögliche Ausführungszeit verursacht. Dies soll sicherstellen, dass die Ausführungszeit in keinem Fall unterschätzt wird. Ein solches Vorgehen ist notwendig, da die WCET-Analyse sonst Gefahr läuft zu optimistisch zu sein. Das gilt auch für das Einführen von *Alibivariablen*, um die Variablen aus vorhergehenden Basisblöcken ersetzen. Im schlimmsten Fall muss eine Variable aus dem Speicher geholt werden. Es ist also notwendig sie aus einer globalen Variable zu laden, um einen möglichen Speicherzugriff zu simulieren. Gleiches gilt auch für die `PHI`-Instruktion am Anfang des Codestückes.

Durch diesen teilweise sehr großen Overhead, wächst dabei die Ausführungszeit des so modifizierten Basisblocks sehr stark im Vergleich zur Ausführungszeit bei normaler Übersetzung

```
1  bb17:
2   %indvar = phi i32 [ %indvar.next, %bb9 ], [ 0, %bb16 ]
3   %tmp = sub i32 0, %b
4   %tmp29 = mul i32 %indvar, %tmp
5   %a.0 = add i32 %tmp29, %a
6   %tmp7 = icmp sgt i32 %a.0, %b
7   br i1 %tmp7, label %bb9, label %bb13.split
```

Abbildung 3.3: Beispiel für einen Basisblock in LLVM

```
1  @indvar.glob = weak global i32 0
2  @b.glob = weak global i32 0
3  @a.glob = weak global i32 0
4  @as = weak global i32 0
5  @bs = weak global i32 0
6
7  define void @main() nounwind {
8  bb17:
9    %b = load i32* @b.glob
10   %a = load i32* @a.glob
11   %indvar = load i32* @indvar.glob
12   %tmp = sub i32 0, %b
13   %tmp29 = mul i32 %indvar, %tmp
14   %a.0 = add i32 %tmp29, %a
15   %tmp7 = icmp sgt i32 %a.0, %b
16   store i32 %b, i32* @bs
17   store i32 %a, i32* @as
18   br i1 %tmp7, label %bb9, label %bb13.split
19 bb9:
20   br label %end
21 bb13.split:
22   br label %end
23 end:
24   ret void
25 }
```

Abbildung 3.4: Basisblock in minimalem Funktionsrumpf

im Kontext der ursprünglichen Funktion. Dort werden üblicherweise Variablen, auf die häufig zugegriffen wird, in Registern gehalten. Dadurch müssen sie nicht aus dem Speicher geladen werden. Auf Ebene des LLVM-Zwischencodes ist es jedoch nicht möglich festzustellen, welche Variablen auf Register abgebildet werden und so muss man immer vom schlimmsten Fall ausgehen und Speicherzugriffe verwenden.

Wie stark die Zunahme der Ausführungszeit jedoch letztendlich ist, hängt davon ab, wieviele Variablen aus anderen Basisblöcken verwendet werden, da für jede Variable ein Speicherzugriff generiert werden muss. Die einzelnen Speicherzugriffe können sich dabei, je nach Architektur noch bei der Ausführung überlappen und so ist es also nicht möglich, den Overhead des generierten Funktionsrumpfes auf einfache Weise aus der Berechnung der Ausführungszeit herauszurechnen. Eine komplexere Möglichkeit diesen Overhead aus der Berechnung zu entfernen, ist es die Ausführungszeit des zusätzlich generierten Codes alleine zu bestimmen und von der Laufzeit der Basisblockfunktion abzuziehen.

Der Hauptproblempunkt dieses Ansatzes ist jedoch, dass man nicht zuviel abziehen darf, da sonst die Abschätzung der Ausführungszeit zu optimistisch wäre. Die Ausführungszeit des Funktionsrumpfes ohne den eigentlichen Code des Basisblocks, darf also nicht nach oben hin abgeschätzt werden. Für eine Zielplattform, die einfach Instruktionen der Reihe nach ohne Überlappung ausführt, ist dies auch gut zu erreichen. Sobald die Zielarchitektur jedoch über eine Pipeline verfügt oder sogar superskalar ausgelegt ist, wird es sehr schwierig mit Hilfe dieser Strategie aussagekräftige obere Schranken für die Ausführungszeit eines Basisblocks zu ermitteln. Effekte wie das Füllen der Pipeline am Anfang einer Instruktionssequenz und die stärker zum Vorschein kommende Latenz beim Zugriff auf Speichervariablen, da diese nicht durch andere Instruktionen versteckt werden kann, führen jedoch dazu, dass die Ausführungszeit des zusätzlich generierten Codes alleine betrachtet größer ist als sie es im Kontext des Basisblockes eigentlich wäre.

In Abbildung 3.4 ist einmal beispielhaft für den Basisblock aus Abbildung 3.3 eine Funktionserzeugung durchgeführt worden. Dabei ist der Code des ursprünglichen Basisblocks grün hinterlegt. In Zeile 9 und 10 wurde je eine `load`-Anweisung für die beiden Variablen `a` und `b` generiert. Die nachfolgende `PHI`-Instruktion wurde ebenfalls durch ein `load` ersetzt. Die beiden `store` Instruktionen, die eingefügt wurden, dienen dazu, bei Übersetzung des leeren Funktionsrumpfes ohne den Code des Basisblockes zu verhindern, dass die beiden `load` Anweisungen für die Variablen `a` und `b` als toter Code deklariert werden und dann in der Optimierungsphase des Übersetzers entfernt werden.

Dieser Ansatz ist also nur unter großen Zugeständnissen an die Analysequalität durchführbar, da in dieser Arbeit Code für eine komplexere Zielarchitektur generiert werden soll. Verwendet

man Alibifunktionen, um den Overhead der Funktionsgenerierung zu nivellieren, so bekommt man dadurch möglicherweise zu optimistische Ausführungszeiten, was für eine WCET-Analyse nicht tolerierbar ist. Ohne deren Verwendung liegen die Ausführungszeiten für einen Basisblock um einen Faktor zwei bis drei über den Ausführungszeiten des wirklich generierten Assemblercodes.

Name	Basisblock in Funktionsrumpf	Funktionsrumpf ohne Basisblock	nivellierte Ausführungszeit	Direkt generierter Assemblercode
bb	58	46	12	14
bb15	49	16	33	10
bb20	14	0	14	4
bb.outer	46	16	30	4
cond_true	58	16	42	14
entry	37	25	12	16

Tabelle 3.1: WCET für verschiedene Arten der Basisblockerzeugung

In Tabelle 3.1 ist für einen iterativen GGT[1]-Algorithmus die beschriebene Generierung von Funktionen aus den einzelnen Basisblöcken durchgeführt worden. Dargestellt sind die Ergebnisse, die durch eine Analyse der einzelnen Codeteile mit dem aiT-Werkzeug berechnet wurden. Der komplette Code des Beispiels ist als LLVM-Zwischencode im Anhang A.2 zu finden. In der zweiten Spalte ist die WCET für die Funktionen, die aus den jeweiligen Basisblöcken generiert wurden, zu sehen. Die dritte Spalte enthält die WCET der des generierten Funktionsrumpfes, jedoch ohne den eigentlichen Code des Basisblocks. In der vierten Spalte findet sich nun die Differenz aus den Spalten zwei und drei, also die um den Overhead der Funktionsgenerierung verminderte WCET des Basisblocks. Die letzte Spalte dient als Vergleich für die Güte der Abschätzung. Dafür wurde die komplette Funktion nach Assembler übersetzt und dann die einzelnen Statements den Basisblöcken zugeordnet.

Diese Ergebnisse decken zwei Schwächen einer solchen Basisblockgenerierung auf. Für die Basisblöcke `bb` und `entry` ist die berechnete Ausführungszeit des Funktionsrumpfes zu hoch, so dass die nivellierte Ausführungszeit in beiden Fällen kleiner ist als die WCET, die für die Assembleranweisungen des Basisblocks bei Übersetzung der kompletten Funktion berechnet wurde. Dadurch kann nicht mehr sichergestellt werden, dass die WCET-Analyse, die diese Basisblockausführungszeit verwendet, sicher, d.h. nicht zu optimistisch, ist. Bei den übrigen Basisblöcken tritt dagegen das Gegenteil auf: Die Ausführungszeiten werden stark überschätzt, bis zum mehrfachen der eigentlichen WCET. Auf dieser Basis lässt sich also keine verlässliche WCET-Analyse implementieren.

[1] Größter gemeinsamer Teiler

Daher soll hier eine andere Möglichkeit vorgestellt werden, um für die Basisblöcke der LLVM-Repräsentation eine maximale Ausführungszeit zu berechnen, um damit dann eine WCET-Analyse durchführen zu können. Hierbei wird bei der Übersetzung des LLVM-Zwischencodes zu Assemblercode darauf geachtet, dass der Kontrollflussgraph nicht mehr entscheidend verändert werden darf. Das bedeutet, dass z.B. Optimierungen von Schleifen, wie *Loop Unrolling*, *Loop Inversion* oder *Loop-invariant Code Motion* nicht während der Codegenerierung eingesetzt werden können, da diese den Kontrollflussgraphen verändern.

Dadurch kann sichergestellt werden, dass ein Basisblock der LLVM-Zwischensprache in schleifenfreien Assemblercode übersetzt werden kann. Allerdings muss der Kontrollfluss im generierten Assemblercode nicht streng linear sein, sondern es sind auch Verzweigungen möglich. Beispielsweise bietet die LLVM-Zwischensprache mit der `br` Instruktion eine Verzweigungsmöglichkeit zu zwei beliebigen Zielbasisblöcken an. Dies lässt sich mit vielen Prozessorbefehlssätzen nicht immer mit nur einer einzigen Instruktion nachbilden, da dort meist nur ein einzelnes Sprungziel pro Instruktion erlaubt ist. Dadurch kann der Kontrollfluss innerhalb des Basisblocks in der Assemblerrepräsentation eine Baumstruktur aufweisen, was die Analyse jedoch nicht entscheidend verkompliziert, da solche Baumstrukturen immer noch ohne Schleifen auskommen und somit ohne äußere Vorgaben analysiert werden können.

Durch den Verzicht auf Optimierungen ab dem Punkt an dem man die Analyse auf dem LLVM-Zwischencode beginnt, erhält man am Ende der Transformation zu Assemblercode für jeden LLVM-Basisblock eine Assemblerrepräsentation, die strukturell weitestgehend dem LLVM-Zwischencode entspricht. Die Ergebnisse in der letzten Spalte von Tabelle 3.1 wurden auf diese Weise generiert. Rechnet man die WCET der einzelnen Basisblöcke auf die komplette Funktion hoch so ist die WCET der Funktion lediglich um einen Faktor 1 bis 2 schlechter als wenn man den optimierten Assemblercode für die komplette Funktion mit dem aiT Analyzer untersucht. Für das Beispiel des GGT bei zwei Iterationen ergab sich für die basisblockweise Bestimmung eine WCET von 80 Zyklen. Übersetzt man die Funktion mit weitergehenden Optimierungen und analysiert den kompletten erzeugten Assemblercode, so erhält man eine WCET von 48 Zyklen. Dies entspricht einem Faktor von ca. 1,7.

In Anbetracht der Zielsetzung eine WCET-Analyse auf der konzeptionell höheren Ebene des LLVM-Zwischencodes durchzuführen, ist es tolerierbar, wenn die Ergebnisse auf einem Niveau unterhalb des aiT-Analysewerkzeuges liegen. Dieses wird auch zur Bestimmung der Basisblockausführungszeiten verwendet, benutzt aber eine Reihe von Optimierungstechniken, um die Effekte von Caches und Pipelines möglichst gut vorherzusagen. Analysiert man damit dann den gesamten Assemblercode einer Funktion, so sorgen diese Techniken für eine bessere WCET-Abschätzung. Werden dagegen nur einzelne Basisblöcke analysiert, dann ist die Effizienz dieser Optimierungen weniger gut. Da es aber nicht Ziel dieser Arbeit ist eine möglichst gute

Low-Levelanalyse durchzuführen, sind die so erreichten Ergebnisse ausreichend.

3.1.3 Annotationen

Daneben ist die Bestimmung von Grenzen für die Ausführung von Schleifen im Kontrollfluss ebenfalls nicht immer auf einfache Weise nur durch Analyse des Programmcodes möglich. Dies funktioniert nur, wenn die Schleifengrenze derart im Quellcode vorgegeben ist, dass sie mit Hilfe von *Constant-* und *Copypropagation* durch das LLVM-Framework ermittelt werden kann. Ist dies nicht der Fall, so müssen sie vom Programmierer vorgegeben werden. Dies geschieht zweckmäßigerweise mit Hilfe von *Annotationen* im Quellcode des Programmes. Auch das Problem mit Rekursionen, wie in Abschnitt 2.3.1 beschrieben, lässt sich durch den Einsatz von Annotationen, welche die maximale Rekursionstiefe vorgeben, lösen.

Annotationen sind Informationen über das Programm, die über das hinausgehen, was durch die verwendete Programmiersprache beschrieben werden kann. Im Bereich von Echtzeitanwendungen hat der Entwickler einer Anwendung meist eine relativ genaue Vorstellung vom zeitlichen Verhalten seiner Anwendung und mit Hilfe von Annotationen wird es ihm ermöglicht, das zeitliche Verhalten von Schleifenkonstrukten und rekursiven Funktionen über ein in der jeweiligen Quellsprache mögliches Maß hinaus zu beschreiben. Durch Annotationen wird dem Programmierer ein Werkzeug an die Hand gegeben, um sein Programm an den Stellen mit Informationen anzureichern, wo diese für eine WCET-Analyse fehlen.

Dabei gibt es zunächst einmal zwei grundlegend verschiedene Ansätze zur Bereitstellung von Annotationen: Eine Möglichkeit wäre es, die zusätzlichen Informationen in einer separaten Datei parallel zum Quellcode unterzubringen und diese der WCET-Analyse zur Verfügung zu stellen. Dies hat jedoch bei der Wartung von Programmcode den Nachteil, dass bei einer Veränderung des Programmquellcodes die Annotationen separat angepasst werden müssen. Dieses Vorgehen ist potentiell sehr fehlerträchtig, da die Annotationen vom Quellcode getrennt sind. Darüberhinaus verkompliziert sich dadurch die Analyse, da sich die Annotationen auf den Quellcode des Programmes beziehen, die Analyse jedoch auf Ebene des LLVM-Zwischencodes durchgeführt werden soll. Man müsste also dann zuerst bestimmen auf welche Stellen im Zwischencode sich die Annotationen beziehen. Dies ist jedoch nicht wünschenswert, da es sehr aufwendig ist, diese Abbildung der Annotationen auf den Zwischencode zu bestimmen. Aus diesen Gründen sind direkt in den Quellcode eingebettete Annotationen vorzuziehen. Diese lassen sich mit Hilfe von verschiedenen Ansätzen realisieren:

- Die Annotationen können innerhalb von Kommentaren der Quellsprache vorgenommen werden. Dieses Vorgehen hat jedoch einige Nachteile. Kommentare werden im Laufe des

Übersetzungsprozesses vom Frontend des Compilers aus dem Code entfernt und tauchen somit nicht mehr in der LLVM-Zwischensprache auf. Dies ließe sich nur durch eine Anpassung des verwendeten Sprachfrontends erreichen. Alternativ dazu könnte man dem Analysewerkzeug den LLVM-Zwischencode und den Quellcode des Programmes als Eingabe mitgeben. Dabei muss dieses jedoch die Strukturen der Quellsprache auf die Struktur der LLVM-Repräsentation abbilden, was wiederum stark vom Compilerfrontend abhängig ist.

- Eine weitere Möglichkeit wäre es, den Umfang der Quellsprache um weitere Schlüsselwörter zu erweitern. Dazu ist es jedoch wiederum notwendig das Frontend für die jeweilige Sprache anzupassen, um die zusätzlichen Informationen in die LLVM-Zwischensprache einzubetten.
- Eine vom Frontend unabhängige Methode um Annotationen bereitzustellen ist es, Funktionsaufrufe von leeren Funktionen zu verwenden, deren Parameter die zusätzlichen Informationen zur Programmbeschreibung enthalten. Diese Aufrufe werden dann vom Frontend des Compilers automatisch mit in die Zwischendarstellung übertragen. Dort können sie dann ohne größere Probleme vom Analysewerkzeug ausgewertet und die enthaltenen Zeitinformationen in die Analyse mit einbezogen werden. [8, Kapitel 2.3]

Die letztere Methode hat gegenüber den beiden vorher genannten einige Vorteile. Man muss dem Analysewerkzeug weder das Quellprogramm zugänglich machen, noch braucht man eine Anpassung des Übersetzerfrontends, um die Annotationen dem Analysewerkzeug bekannt zu machen. Auch die Quellsprache muss nicht an die WCET-Analyse angepasst werden. Durch geeignete Spezifikation der verwendeten Funktionen lassen sich auch Annotationen aus mehreren Quellsprachen einheitlich für die Analyse nutzen. Dies sorgt dafür, dass die Analyse selbst ebenfalls quellsprachenunabhängig bleibt, da diese so alle von ihr benötigten Eingabedaten über den LLVM-Zwischencode der analysierten Funktionen erhält.

3.1.4 Behandlung von rekursiven Funktionen

Ein weiteres Problem bei der Bestimmung der WCET stellen Rekursionen dar. Bei einfachen Funktionsaufrufen genügt es, die WCET des aufrufenden Basisblocks um die WCET der aufgerufenen Funktion zu erhöhen. Bei rekursiven Funktionen gestaltet sich dies komplexer. Im einfachsten Fall ruft sich eine rekursive Funktion nur selbst auf. Ist die maximal mögliche Rekursionstiefe bekannt, so ist die WCET eines Aufrufs dieser Funktion die WCET einer einmaligen Abarbeitung multipliziert mit der Rekursionstiefe.

Rekursionen können sich jedoch auch über mehrere Aufrufe hinweg fortsetzen. Dies erkennt man durch Zyklen im Aufrufgraph des betrachteten Programmes. In diesem Fall benötigt man die WCET aller in einem Aufrufzyklus enthaltenen Funktionen, um die Ausführungszeit eines Aufrufes nach oben hin abschätzen zu können. Die maximale Rekursionstiefe gibt in diesem Fall die maximale Anzahl der Ausführungen des Aufrufzykluses an.

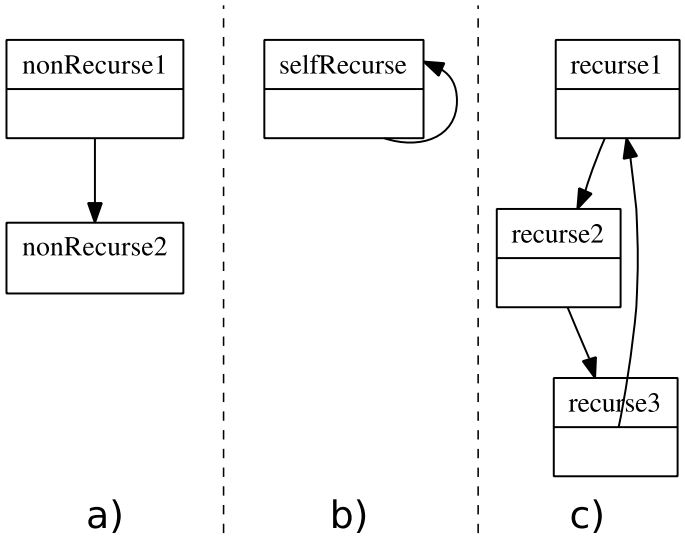


Abbildung 3.5: Beispiele für die verschiedenen Aufruftypen

In Abbildung 3.5 ist der Aufrufgraph eines Moduls mit 6 Funktionen zu sehen. In Teil a) sind die beiden Funktionen `nonRecurse1` und `nonRecurse2` abgebildet. `nonRecurse1` enthält dabei einen Aufruf von `nonRecurse2`, die ihrerseits jedoch keine weiteren Funktionen aufruft. In diesem Fall liegt also keine Rekursion vor. Die Ausführungszeit des Basisblocks in `nonRecurse1`, in dem der Aufruf stattfindet wird dabei durch die WCET der aufgerufenen Funktion `nonRecurse2` erhöht.

In Teil b) sieht man die Funktion `selfRecurse`, die sich selbst aufruft. Dies ist die einfachst mögliche Art der Rekursion. Die maximale Ausführungszeit eines Basisblocks, in dem eine solche Funktion aufgerufen wird, verlängert sich dabei für jede Rekursionsstufe einmal um die maximale Ausführungszeit der rekursiven Funktion, also um Rekursionstiefe mal Ausführungszeit der Funktion.

In c) ist ein Aufrufzyklus bestehend aus drei Funktionen `recurse1`, `recurse2` und `recurse3` dargestellt. In diesem Fall gibt die maximale Rekursionstiefe die Anzahl komplett ausgeführter Zyklen an.

In vielen Frameworks zur WCET Analyse wird das Thema Rekursion komplett ausgespart. So wird in dem in Kapitel 5.4.3 vorgestellten Framework zur WCET-Bestimmung von C-Code

explizit der rekursive Aufruf von Funktionen ausgeschlossen. Die Tiefe einer Rekursion lässt sich schlecht automatisch abschätzen und eine durch Annotationen vorgegebene Rekursionstiefe lässt sich im Fall von komplizierteren Zyklen im Aufrufgraphen nur schlecht interpretieren. Deswegen werden hier nur einfache Zyklen im Aufrufgraph unterstützt. Die hier gewählte Form der Realisierung unterstützt somit zwar nicht alle Formen von Rekursion, aber die häufigsten Arten werden berücksichtigt und können mit Hilfe von Annotationen in der Rekursionstiefe beschränkt werden.

3.2 Überblick über die Implementierung

Nachdem nun einige zentrale Fragen einer WCET-Analyse auf LLVM-Zwischensprachenebene näher beleuchtet wurden, soll jetzt im folgenden Abschnitt konkret der Aufbau und die Funktionsweise der Implementierung vorgestellt und erläutert werden. Die Analyse erfolgt, wie eben beschrieben, auf dem LLVM-Zwischencode. Als Zielarchitektur wird ein *PowerPC MPC565* verwendet. Die Codegenerierung wird dabei vom generischen PowerPC Backend der LLVM übernommen, welches, wie in Kapitel 3.1.2 beschrieben angepasst wurde, um Code für die jeweiligen Basisblöcke zu erhalten. Der generierte Assemblercode wird mit Hilfe der aiT Version für ebendiesen Prozessor analysiert.

3.2.1 Aufbau der Implementierung

Die Implementierung der WCET-Analyse nach dem in Kapitel 3.1.1 vorgestellen Ansatz hat dabei die in Abbildung 3.6 vorgestellte Struktur.

Als Eingabe dient der LLVM-Zwischencode, der in Form eines Moduls mit allen zu analysierenden Funktionen dem `PassManager` übergeben wird. Dieser führt dann im weiteren Verlauf die verschiedenen Aktionen, die als Passes vorliegen, aus. Die einzelnen Passes der WCET-Analyse können über globale Datenstrukturen miteinander kommunizieren. So existiert für jede Funktion eine Struktur namens `WCETData`. Diese enthält den `T-Graph` (vgl. Kapitel 2.1), der aus dem LLVM-Zwischencode der Funktion generiert wird, den dazu passenden `LPSolver`, der das dem `T-Graph` entsprechende LP-Problem kapselt und eine `CallMap`, die für jeden Basisblock die enthaltenen Funktionsaufrufe und Aufruftypen speichert. Davon zunächst getrennt werden die Iterationsgrenzen für Schleifen und die Rekursionstiefe für rekursive Funktionsaufrufe in den assoziativen Datenstrukturen `LoopBoundMap` und `RecursionBoundMap` gespeichert.

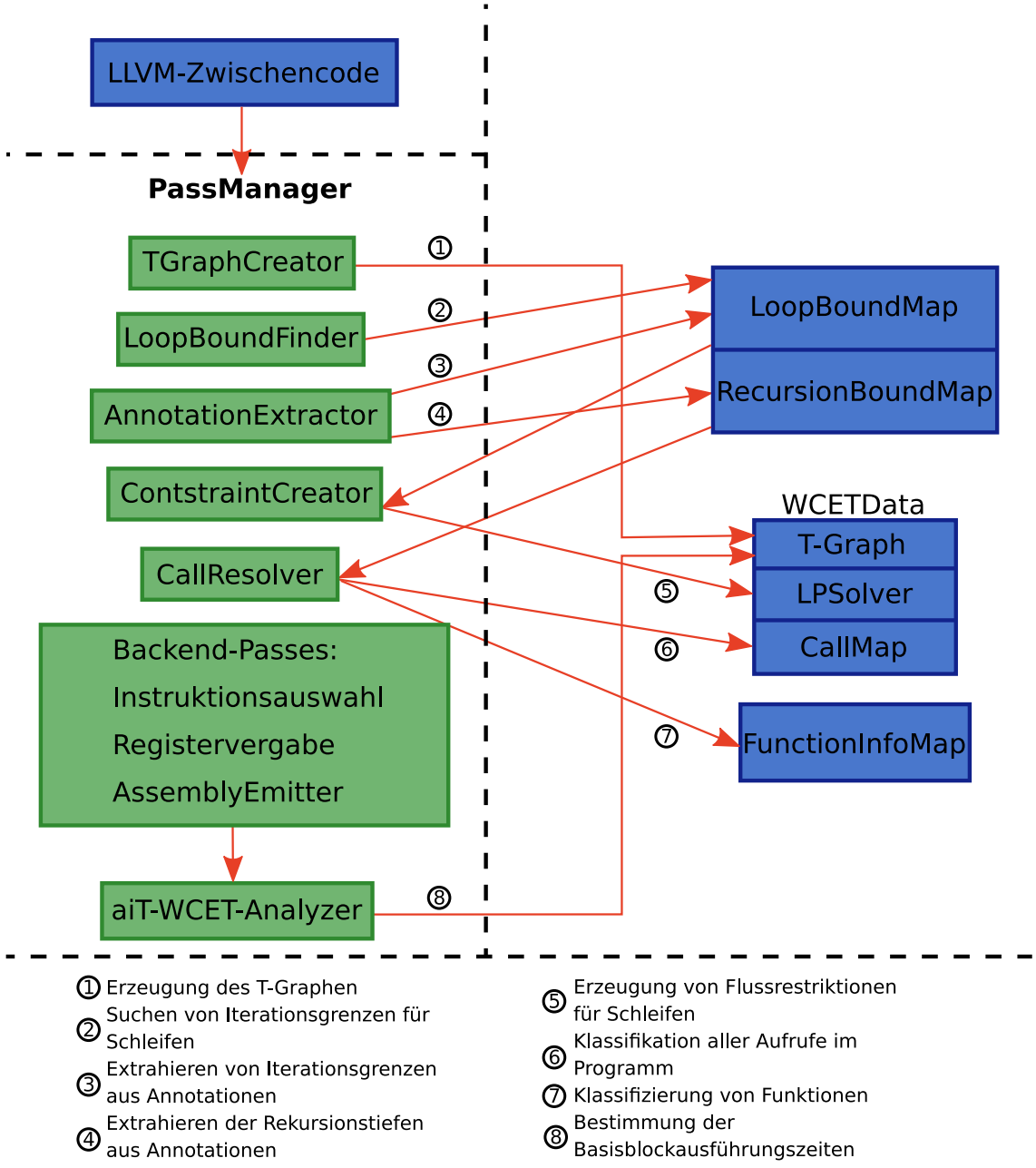


Abbildung 3.6: Aufbau der Implementierung

Die Verarbeitung beginnt zunächst mit der Erzeugung eines `T-Graphen` für jede Funktion innerhalb des analysierten Moduls. Diese Aufgabe übernimmt der `TGraphCreator`. Das Ergebnis wird in der zur Funktion gehörigen `WCETData` Struktur gespeichert. Danach wird versucht mit Hilfe des LLVM-Frameworks mit dem `LoopBoundFinder` obere Grenzen für die Anzahl der Ausführungen von Schleifen zu finden. Daraufhin werden eventuell vorhandene Annotationen im zu übersetzenden Code vom `AnnotationExtractor` ausgewertet und anschließend

aus dem Programm entfernt. Mit Hilfe solcher Annotationen kann der Programmierer Grenzen für die Ausführung von Schleifen und die Rekursionstiefe einzelner Funktionsaufrufe angeben. Die mit dem `LoopBoundFinder` und dem `AnnotationExtractor` gefundenen Iterationsgrenzen für Schleifen werden in getrennten `LoopBoundMaps` gespeichert, die Rekursionstiefen für Funktionsaufrufe werden in einer `RecursionBoundMap` vermerkt. Nun werden für die Schleifen des Programmes aus den entweder automatisch ermittelten oder manuell vorgegebenen Iterationsgrenzen im `ConstraintCreator` Flussrestriktionen(vgl. Kapitel 2.1.4) ermittelt und diese direkt zum `LPSolver` der Funktion hinzugefügt. Der `ConstraintCreator` kann dabei die Informationen aus beiden `LoopBoundMaps` auslesen und somit auswählen, welchen der beiden er den Vorzug gibt. Als nächstes läuft der `CallResolver`, der feststellt ob eine Funktion rekursiv ist oder nicht und dies in der `FunctionInfoMap` festhält. Darüberhinaus wird für jeden Aufruf einer Funktion in der `CallMap` festgehalten zu welchem Basisblock dieser Aufruf gehört gehört. Außerdem wird dort festgehalten, wie oft die Funktion aufgerufen wird. Bei normalen Funktionsaufrufen ist dies üblicherweise einmal der Fall. Bei rekursiven Funktionen wird hier die annotierte Rekursionstiefe eingetragen, die dann angibt, wie oft der zugehörige Aufrufzyklus ausgeführt wird.

An dieser Stelle der Verarbeitung schließt sich dann das LLVM-Backend an, das den LLVM-Zwischencode in die Assemblersprache des Zielprozessors mit den im LLVM-Framework vorhandenen Methoden übersetzt. Am Ende dieses Übersetzungsvorgangs existiert dann für jeden Basisblock im LLVM Kontrollflussgraphen eine Assemblerrepräsentation, für die dann mit Hilfe des aiT-WCET-Analysewerkzeuges (siehe Kapitel 2.4) die maximale Ausführungszeit bestimmt werden kann. Dies lässt sich weitestgehend ohne Annotationen im erzeugten Assemblercode bewerkstelligen, da komplizierte Kontrollstrukturen wie z.B. Schleifen nur im Kontrollflussgraphen der LLVM-Funktion auftauchen, die Analyseeinheiten jedoch nur einzelne Basisblöcke davon umfassen. Damit die Kontrollflussstruktur möglichst erhalten bleibt, werden im Backend des LLVM-Compilers nur die nötigsten Operationen ausgeführt. Dazu gehören u.a. Instruktionsauswahl, Registerallokation und Instruktionsablaufplanung. Wird das Eingabeprogramm nun für die Ausführung auf dem Zielsystem mit einer erweiterten Menge von Optimierungen übersetzt, so muss man davon ausgehen können, dass diese Optimierungen die Ausführungszeit immer verbessern oder unverändert lassen verglichen mit der Ausführungszeit bei Übersetzung mit einer minimalen Menge an Operationen zur Codeerzeugung. Dies lässt sich jedoch mit einem Übersetzer auf Basis des LLVM Frameworks relativ einfach realisieren, da man dort die einzelnen Optimierungen einzeln aktivieren und somit problematische Operationen bei der Übersetzung verhindern kann. Die nun bekannte Ausführungszeit der einzelnen Basisblöcke bildet die Grundlage für die Kantengewichte im `T-Graphen` und wird in diesen vermerkt.

Ist dieser Teil der Analyse abgeschlossen, so liegt nun für jede Funktion des Übersetzungsmoduls ein `T-Graph`, ein `LPSolver` und eine `CallMap` mit allen notwendigen Informationen über das zeitliche Verhalten des analysierten Programmes vor. Zur endgültigen Berechnung der WCET müssen nun noch die Ausführungszeiten der Basisblöcke, in denen Funktionsaufrufe auftreten, mit Hilfe der in `CallMap` enthaltenen Informationen angepasst werden, da die Ausführungszeit, die in den aufgerufenen Funktionen verbracht wird, ebenfalls zur Ausführungszeit des Basisblocks zählt. Dann folgt die Konversion des `T-Graphen` in ein LP-Problem, welches man mit dem `LPSolver` lösen kann und so die WCET erhält.

Die Möglichkeit die Daten über die in LLVM vorhandenen Mechanismen von Pass zu Pass weiterzureichen wird nicht verwendet, weil sich der `AnnotationExtractor` nicht in das Konzept von wiedererzeugbaren Informationen einfügen lässt. Die LLVM verlässt sich jedoch darauf, dass Informationen, die durch die Ausführung von weiteren Passes vernichtet wurden, wieder erzeugbar sind, indem der zugehörige Pass nochmals ausgeführt wird. Im Fall von typischen Vorgängen innerhalb eines Übersetzers ist dies durchaus zutreffend. So ist es z.B. möglich Informationen über die Schleifen in Programmen wieder neu zu erzeugen. Problematisch ist auch, dass hier Daten über das Programm aus verschiedensten Ebenen an zentraler Stelle, nämlich dem T-Graphen, zusammentreffen. Dieser enthält Daten aus konzeptionell höherliegenden Schichten, wie z.B. die durch Annotationen angegebenen Iterationsgrenzen für Schleifen, sowie auch aus niedrigeren Schichten, wie z.B. die Basisblockausführungszeiten, die aus dem vom Backend generierten Assemblercode berechnet werden.

Die einzelnen Berechnungsschritte der Analyse sind dabei in LLVM-Passes aufgeteilt und fügen sich so nahtlos in den Übersetzungsprozess des LLVM-Frameworks ein. Sie können somit einfach zu einem `PassManager` hinzugefügt werden, um dann analog zu den einzelnen `Passes` des Backends abgearbeitet zu werden. Im Anschluss folgt nun eine Beschreibung der einzelnen Passes, die für die Durchführung der WCET-Analyse verwendet werden.

3.2.2 Erzeugung des T-Graphen

Am Anfang des Analyseprozesses steht die Erzeugung der T-Graphstruktur. Für jede Funktion des Übersetzungsmoduls wird dabei ein dazugehöriger T-Graph generiert. Daher bietet es sich an, diesen Prozess in einem `FunctionPass` durchzuführen, so dass die Generierung von LLVM automatisch für jede Funktion eines Übersetzungsmoduls durchgeführt wird. Die Ergebnisse der T-Graphgenerierung werden dann in einer globalen Datenstruktur gespeichert, so dass sie von späteren Stufen der Analyse auf einfache Weise verwendet werden können. Die Kanten des T-Graphen entsprechen dabei den Basisblöcken der Funktion. Zusätzlich wird jedoch für jede Verzweigung im Kontrollfluss ein Paar weitere Kanten erzeugt. Dies ist notwendig, da sich der

verzweigende Kontrollfluss nicht mit Hilfe von einer einzigen Kante beschreiben lässt, da diese nur auf einen einzelnen Nachfolger zeigen könnte. Den eingefügten Verzweigungskanten wird dabei ein Kantengewicht von null zugeordnet, so dass für die WCET Berechnung lediglich das Gewicht der vorangegangenen Kante entscheidend ist. Dieses ist durch die WCET des jeweils zugehörigen Basisblocks gegeben. Die WCET der Basisblöcke steht jedoch erst am Ende des Übersetzungsvorgangs zur Verfügung, so dass der T-Graph ohne Kantengewichte initialisiert wird und somit zunächst noch unvollständig ist.

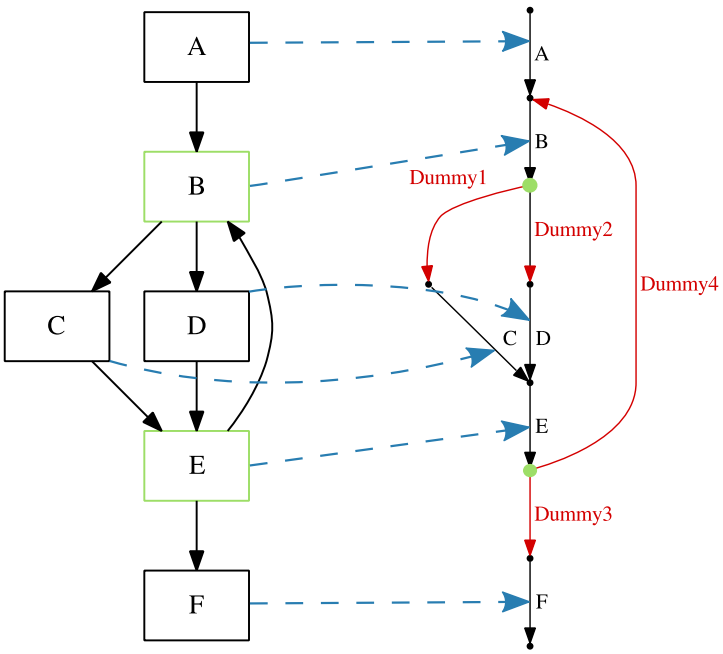


Abbildung 3.7: Beispiel eines Kontrollflussgraphen und der dazu generierte T-Graph

In Abbildung 3.7 ist die Transformation von Kontrollflussgraph (links) zu T-Graph (rechts) nochmals verdeutlicht. Man sieht die Abbildung von Knoten des Kontrollflussgraphen auf Kanten im T-Graph. Die im Kontrollflussgraphen grün gekennzeichneten Knoten besitzen mehr als einen Nachfolger. Im T-Graph sieht man an den entsprechenden, ebenfalls grün markierten Stellen, dass dort für die Verzweigung des Kontrollflusses die rot hervorgehobenen Hilfskanten `Dummy1` bis `Dummy4` eingefügt wurden. Die Verzweigungsstruktur ist damit äquivalent zur Struktur des Kontrollflussgraphen und ein Block im Kontrollflussgraph kann so beliebig viele Verzweigungen zu anderen Blöcken aufweisen.

3.2.3 Automatisches Bestimmen von Schleifengrenzen

Nachdem die Struktur des T-Graphen im vorhergehenden Pass erstellt wurde, wird dieser nun in einem ersten Schritt stufenweise um Informationen ergänzt, damit er am Ende dieses Prozesses

die zeitlichen Abläufe des Eingabeprogrammes genau beschreibt. Eine wichtige Voraussetzung, um die WCET mit Hilfe von Zirkulationen auf dem T-Graph zu bestimmen, ist die Ermittlung von Grenzen für die Ausführung von Schleifen. Wie in Kapitel 3.1.3 vorgestellt, kann man diese mit Hilfe von Annotationen vom Programmierer vorgeben lassen. Bei einfachen Schleifen jedoch, für welche die Schleifengrenzen schon aus dem Quellcode selbst ersichtlich sind, ist es möglich, mit Hilfe des LLVM-Frameworks diese direkt im LLVM-Zwischencode zu bestimmen. So wird dem Programmierer die Arbeit erleichtert, weil er nicht jede Schleife extra annotieren muss, und die Fehlerquelle falsch angegebener Annotationen wird minimiert.

Diese Funktionalität wird vom `LoopBoundFinder` zur Verfügung gestellt. Dieser ist als `FunctionPass` implementiert und iteriert über alle Schleifen einer Funktion. Für Schleifen, die in Normalform[2] vorliegen, kann die LLVM die Anzahl der Schleifendurchläufe als LLVM-Ausdruck angeben. Dieser muss jedoch nicht zwingenderweise konstant sein, sondern kann auch eine Operation aus beliebigen Variablen des Programm sein. Nur für den Fall, dass dieser Ausdruck eine Konstante ist, kann er als Schleifengrenze verwendet werden. Trifft dies zu, so speichert die Implementierung den ermittelten Wert in der `LoopBoundMap` ab, so dass er zu einem späteren Zeitpunkt von weiteren Teilen der Analyse verwendet werden kann.

3.2.4 Auswerten von Annotationen

Annotationen werden verwendet, um dem Programmierer die Möglichkeit zu geben weitergehende Informationen für die WCET-Analyse anzugeben. Sie werden, wie in Kapitel 3.1.3 vorgestellt, durch leere Funktionsaufrufe realisiert, die aus dem LLVM-Zwischencode extrahiert werden können. Dabei existieren zwei grundlegende Arten von Annotationen. Zum einen lassen sich Ausführungsgrenzen für Schleifen annotieren, zum anderen die Rekursionstiefe für rekursive Funktionsaufrufe festlegen.

Manuelles Festlegen von Schleifengrenzen

Kann für eine Schleife mit Hilfe der im vorigen Kapitel vorgestellten Methodik keine Grenze gefunden werden, so muss diese durch Annotationen vorgegeben werden, um eine WCET-Analyse durchführen zu können. Für die Festlegung von Schleifengrenzen stehen folgende in Abbildung 3.8 dargestellten Annotationen bereit.

[2] In diesem Fall bedeutet dies, dass die Schleifenabbruchbedingung durch einen Ganzzahlzähler dargestellt wird, der von Null ab bei jedem Schleifendurchlauf nach oben gezählt wird und bei Erreichen eines bestimmten Wertes abbricht.

```
void setLoopCount(int times);
void setVariableLoopCount(int variableBound, int hardBound);
```

Abbildung 3.8: Annotationen für Schleifengrenzen

Mit Hilfe der ersten Version lässt sich eine Grenze für eine Schleife fest vorgeben. Der verwendete Parameter muss eine Konstante sein. Ist dies nicht der Fall, so meldet die Analyse einen Fehler beim Auswerten der Annotation. Damit lässt sich jedoch lediglich eine starre Grenze für die Anzahl der Schleifenausführungen vorgeben. Etwas mehr Freiheit gewinnt man durch Verwendung der zweiten Version. Damit ist es möglich, einen beliebigen Ausdruck in der Quellsprache als ersten Parameter anzugeben. Kann dieser im Laufe des Übersetzungsprozesses vom LLVM-Compiler zu einer Konstante evaluiert werden, so wird dieser Parameter als Grenze für die Schleifenausführung verwendet, ansonsten wird auf den zweiten Parameter zurückgegriffen, der auf jeden Fall, analog zum Parameter der ersten Annotationsfunktion, immer konstant sein muss.

```
for (int i = 0; i < bound; i += 2) {
    setVariableLoopCount(bound / 2 + 1, 10);
    ...
}
```

Abbildung 3.9: Komplexe Annotation

In Abbildung 3.9 ist eine Anwendung einer solchen komplexen Annotation zu sehen. Kann der Wert der Variable `bound` zur Übersetzungszeit bestimmt werden, so evaluiert der Ausdruck `bound / 2 + 1` zu einer Konstante und wird so vom Analysewerkzeug als Schleifengrenze verwendet. Ist dies nicht der Fall, so kommt der zweite Parameter zum Zug. Hieran sieht man auch einen Vorteil, der sich bei Modifikation des Quellcodes ergibt: Wird der Wert von `bound` im Laufe der Entwicklung verändert, so muss die Annotation nicht angepasst werden. Die Wartbarkeit des Codes wird also verbessert. Wie man ebenfalls sieht, wird die Annotation innerhalb des Schleifenrumpfes notiert. Dies vereinfacht die Zuordnung der Annotation zur dazugehörigen Schleife, da so sichergestellt wird, dass der Übersetzer bei der Generierung des LLVM-Zwischencodes die Annotation nicht beliebig verschieben kann. Es ist auch einfacher festzustellen, zu welcher Schleife die Annotation gehört, da die Annotation sich innerhalb der Schleife befindet. Dies ist insbesondere bei verschachtelten Schleifen vorteilhaft.

Mit Hilfe dieser Annotationen müssen alle Schleifen, deren Grenzen die LLVM nicht selbstständig bestimmen kann, mit einer Ausführungsgrenze versehen werden, um eine WCET-Analyse zu ermöglichen.

Festlegen der Rekursionstiefe

Analog zu den Annotationen für Schleifengrenzen existieren auch Annotationen für die Angabe der Rekursionstiefe. Diese kann nicht mit Hilfe der LLVM bestimmt werden und muss so in jedem Fall vom Programmierer der Anwendung vorgegeben werden. Die Annotationen sind dabei ähnlich aufgebaut wie die für Schleifengrenzen:

```
void setRecursionDepth(int times);
void setVariableRecursionDepth(int variableBound, int hardBound);
```

Abbildung 3.10: Annotationen für Rekursionstiefe

Die erste Version dient dabei wieder dazu, einen festen Wert für die Rekursionstiefe anzugeben, bei der zweiten Version kann man als ersten Parameter einen variablen Ausdruck angeben, der, falls er zu einer Konstante evaluiert werden kann, als Rekursionstiefe verwendet wird. Die Anwendung erfolgt dabei wie in folgendem Beispiel.

```
int foo() {
    setRecursionDepth(10);
    int f = fak(10);
    return f;
}
```

Abbildung 3.11: Anwendung einer Rekursionsannotation

Die Annotation erfolgt hier also vor dem Aufruf der rekursiven Funktion `fak`. Die Interpretation der Rekursiontiefe erfolgt analog zu dem in Kapitel 3.1.4 angegebenen Verfahren. Für jeden Aufruf einer rekursiven Funktion muss so eine Annotation der Rekursionstiefe erfolgen. Alternativ wäre es auch möglich, die Annotation an die Deklaration oder Implementierung der rekursiven Funktion zu setzen. Das hat aber den Nachteil, dass dann die Rekursionstiefe, die zur Analyse verwendet würde, bei jedem Aufruf gleich wäre. Die hier gewählte Lösung ist also flexibler, da man so die Rekursionstiefe für jeden Aufruf neu an die Eingabeparameter anpassen kann und so die Qualität der Analyse verbessern, da diese dann weniger pessimistisch ausfallen kann.

Implementierung der Annotationserkennung

Sämtliche im Eingabeprogramm auftauchenden Annotationen werden mit dem `Annotation-Extractor` aus dem Programm extrahiert und die in ihnen enthaltenen Informationen über das

Ablaufverhalten des Programmes interpretiert und für die Analyse nutzbar gemacht. Die Implementierung erfolgt als `FunctionPass`. Jede Funktion wird dabei nach den Signaturen der zur Annotation verwendeten Funktionen durchsucht. Die Informationen aus den Funktionsparametern werden daraufhin, wie in den beiden vorhergehenden Abschnitten beschrieben, interpretiert und dann, je nach Art der Annotation, in der entsprechenden `LoopBoundMap` bzw `RecursionBoundMap` abgelegt, so dass nachfolgende Verarbeitungsschritte sie nutzen können. Danach werden die Aufrufe der Annotationsfunktionen aus dem LLVM-Zwischencode entfernt, so dass bei der weiteren Übersetzung kein Overhead mehr für ihren Aufruf existiert.

Probleme der Annotationserkennung

Der Vorteil der auf diese Weise implementierten Annotationen ist, dass sie unabhängig vom Frontend des LLVM-Übersetzers für die Implementierung verständlich sind. Solange man die Annotationen aus dem unoptimierten LLVM-Zwischencode, den das Frontend erzeugt, extrahiert, ist dies auch ohne Einschränkungen gültig. Dieser stellt nämlich eine 1-zu-1 Abbildung des Eingabeprogramms auf den LLVM-Zwischencode dar. Sobald jedoch Optimierungen darauf ausgeführt werden, muss die Struktur des Programmes nicht mehr der Struktur des Quellcodes entsprechen, sondern das optimierte Programm hat dann lediglich dieselben funktionalen Eigenschaften wie die ursprüngliche Version. Dies gilt insbesondere für die Klasse von kontrollflussverändernden Optimierungen. Darunter fallen auch viele Schleifenoptimierungen, wie z.B. *Loop-Unrolling*. Diese sind vor allem für die Annotationen der Schleifen problematisch. Die Annotationen müssten dazu von den entsprechenden Optimierungspasses entsprechend der durchgeführten Operationen ebenfalls transformiert werden, um solche Optimierungen im Kontext der WCET-Analyse weiter benutzen zu können. Wird z.B. eine Schleife ausgerollt, dann kann die Annotation der Anzahl der Schleifenausführungen vollständig überflüssig sein, wenn die Schleife komplett durch sequenziellen Code ersetzt wird. In dem Fall, dass die Schleife nicht komplett ausgerollt wird, sondern lediglich im Schleifenkörper mehrere ursprüngliche Iterationen durch eine einzige ersetzt werden, dann muss der Wert der Annotation entsprechend auf die niedrigere Ausführungsanzahl angepasst werden. Analog gilt dies für rekursive Funktionen, deren Aufruf von Optimierungspasses eingebettet wird. Bei rekursiven Funktionen ist dies jedoch eher unüblich, da der Optimierer dazu die Tiefe der Rekursion ermitteln müsste, und diese nicht zu groß sein darf, damit sich diese Optimierung lohnt. Dies kann man also näherungsweise vernachlässigen, da dieser Fall eher unwahrscheinlich ist.

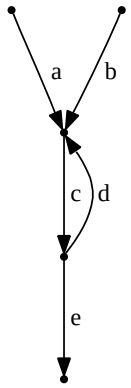
Das Problem mit den Schleifenannotationen ist jedoch von stärkerem Belang, da derartige Optimierungen sehr oft zum Einsatz kommen. In der aktuellen Implementierung wird jedoch davon

ausgegangen, dass die Struktur des Programmes nicht derart verändert wird, dass die Bedeutung der Annotationen nicht mehr richtig zugeordnet werden kann.

3.2.5 Erzeugung von Flussrestriktionen für Schleifen

An diesem Punkt der Analyse sollte nun für jede Schleife die maximale Anzahl der Ausführungen bekannt sein. Entweder kann diese durch die LLVM bestimmt werden oder der Programmierer gibt sie mit Hilfe von Annotationen vor. Daraus werden nun Restriktionen für den Fluss durch den T-Graphen generiert (siehe Kapitel 2.1.4). Diese Aufgabe wird durch den `LoopConstraintCreator` ausgeführt, der als `LoopPass` implementiert ist. Für jede Schleife wird dabei geprüft, ob die Schleifengrenze automatisch oder per Annotation manuell spezifiziert wurde. Ist beides der Fall, so ist es sinnvoll das Minimum der beiden Werte zu nehmen.

Die Generierung von Flussrestriktionen aus den Schleifengrenzen soll nun im Folgenden an einem Beispiel demonstriert werden.



Die Schleife (c,d) soll maximal fünf mal ausgeführt werden. Der Fluss durch die Kanten c und d ist also kleiner gleich 5:
$f(c) \leq 5; f(d) \leq 5$

Abbildung 3.12: Ausschnitt aus einem T-Graph mit Schleife

Auffällig an diesem Beispiel ist, dass die Schleife (c,d) zwei eingehende Kanten besitzt. Mit Hilfe der in Kapitel 2.1.4 beschriebenen Flussrestriktionen soll nun beschrieben werden, dass die Schleife maximal fünf mal ausgeführt werden soll. Dazu muss man den in die Schleife eingehenden Fluss mit dem Fluss innerhalb der Schleife in Beziehung setzen. Zuerst werden alle Vorgänger der Schleife gesucht, um den in die Schleife eingehenden Fluss zu ermitteln. Im hier betrachteten Beispiel sind dies die Kanten a und b. Der eingehende Fluss beträgt also $f(a) + f(b)$. Ist, wie im Beispiel angenommen, die Grenze für die Anzahl der Schleifenausführungen fünf, so wird für jede Ausführung eines Vorgängers der Schleife diese maximal fünf mal abgearbeitet. Der maximale Fluss durch die Kante c ist also fünf mal so groß wie der gesamte eingehende Fluss:

$$5(f(a)+f(b)) \leq f(c)$$

Dies ist äquivalent zur folgenden Ungleichung, die sich in der für ein LP-Problem tauglichen Form befindet:

$$5f(a)+5f(b) \leq f(c)$$

Es genügt dabei, den Fluss einer Kante der Schleife mit ihren Vorgängern in Beziehung zu setzen, da durch die Flusserhaltung implizit alle Kanten der Schleife, ausgenommen der Rückwärtskante, denselben Fluss aufweisen. Deren Fluss ist um eins kleiner als der Fluss der restlichen Kanten in der Schleife, da bei der letzten Iteration die Schleife verlassen, also die Rückwärtskante nicht genommen wird.

Die erzeugten Restriktionen werden dann direkt dem zur betrachteten Funktion gehörigen LP-Problem hinzugefügt.

3.2.6 Klassifizierung von Funktionsaufrufen

Im nun folgenden Schritt wird der Aufrufgraph des zu analysierenden Programmes untersucht. Dabei wird festgestellt, welche Funktionen rekursives Verhalten aufweisen. Diese Information wird benötigt, um die endgültige Ausführungszeit von Basisblöcken zu berechnen, in denen Funktionsaufrufe vorhanden sind. Diese setzt sich zusammen aus der Ausführungszeit des für den Basisblock generierten Assemblercodes und der Ausführungszeit der aufgerufenen Funktionen. Der Aufrufgraph enthält für jede Funktion eines Übersetzungsmoduls die Information welche anderen Funktionen von ihr aufgerufen werden.

Der `CallResolver` traversiert den Aufrufgraph und kann dann feststellen, welche Funktionen sich in einem Aufrufzyklus befinden. Es werden drei verschiedene Arten von Funktionsaufrufen erkannt:

- Nicht rekursive Funktionsaufrufe
- Einfache rekursive Funktionen, die sich direkt selbst aufrufen
- Komplexere Aufrufzyklen aus mehreren Funktionen

Tauchen komplexere Strukturen im Aufrufgraphen auf, wie z.B. zwei Schleifen, die sich einige Knoten teilen, dann wird ein Fehler signalisiert. Die Implementierung kann also nur einfache, sich nicht überschneidende Zyklen im Aufrufgraphen behandeln.

Die Grenzen für die maximale Rekursionstiefe müssen der Analyse mit Hilfe von Annotationen bekannt gemacht werden. Der `CallResolver` muss also nach dem Extrahieren und Auswerten der Annotationen ausgeführt werden, damit er auf diese Informationen zugreifen kann.

Diese hier beschriebene Analyse funktioniert jedoch lediglich für direkte Aufrufe im Programm. Sobald Funktionszeiger verwendet werden, ist es nicht mehr möglich auf Ebene des LLVM-Zwischencodes sicher festzustellen welche Funktionen vom Programm aufgerufen werden. Dies ist allein deshalb unpraktikabel, da man dann Wissen über das fertig nach Objektcode übersetzte und gebundene Programm besitzen müsste, um den Wert der Funktionszeiger als Aufruf einer konkreten Funktion interpretieren zu können. Diese Anforderung jedoch läuft dem Ansatz, die Analyse auf Ebene der LLVM-Zwischensprache durchzuführen, entgegen und wurde deshalb nicht weiter verfolgt.

In welche Kategorie die Funktion einsortiert wird, speichert der `CallResolver` in der `FunctionInfoMap`. Für jeden Funktionsaufruf innerhalb des analysierten Programmes wird außerdem noch in der zur aufrufenden Funktion gehörigen `CallMap` vermerkt, in welchem Basisblock dieser stattfindet. Damit kann im weiteren Verlauf der Analyse auf einfache Weise der Einfluss des Aufrufs auf die WCET des Basisblocks, in dem der Aufruf stattfindet, bestimmt werden.

3.2.7 Ausführungszeiten von Basisblöcken

Ist dieser Schritt ausgeführt, dann sind fast alle Informationen, welche die WCET-Analyse benötigt, vorhanden. Es fehlen lediglich die Ausführungszeiten der Basisblöcke. Diese sollen nun im nächsten Schritt bestimmt werden. Dazu wird der LLVM-Zwischencode zunächst mit Hilfe der im LLVM-Framework vorhandenen Werkzeuge in Assemblercode umgewandelt. Dieser wird dann vom aiT-Analyzer untersucht und seine WCET bestimmt.

Assemblergenerierung

Das Ziel ist es, bei dieser Übersetzung für jeden Basisblock einzeln Assemblercode zu erhalten. Darum ist es notwendig den Kontrollflussgraphen möglichst nicht zu verändern. Es dürfen demnach keine Kontrollflussgraphtransformationen stattfinden. Das ist notwendig, damit man für einen Basisblock auf LLVM-Zwischencodeebene möglichst ein sequenzielles Stück Assemblercode erhält. Zur Codegenerierung werden deshalb nur die wirklich notwendigen Passes verwendet. Wird für den auf dem Zielsystem ausgelieferten Code eine höhere Optimierungsstufe verwendet, so wird davon ausgegangen, dass diese die WCET nicht verschlechtert, sondern

mindestens unverändert lässt. Am Ende wird dann der Assemblercode für jeden einzelnen Basisblock der LLVM-Zwischensprache in eine eigene Datei geschrieben. Diese wird assembliert, so dass sie vom aiT-Analyzer verarbeitet werden kann.

Die Erzeugung von maschinenabhängigem Code erfolgt grob in folgenden Schritten: Instruktionsauswahl, Registervergabe und Instruktionsablaufplanung. Dazu werden die zielplattformspezifischen Passes der LLVM verwendet, um diese Schritte nacheinander durchzuführen. Ist dies geschehen, dann liegt eine maschinenspezifische Beschreibung des Programmes als Datenstruktur im Speicher. Diese muss dann noch in das gewünschte Ausgabeformat, in diesem Fall Assembler, transformiert werden. Das ist allerdings an dieser Stelle lediglich eine 1 zu 1 Abbildung der Datenstruktur im Speicher. Die bis zu diesem Zeitpunkt der Codegenerierung verwendeten Passes sind alles Teile des plattformspezifischen LLVM-Backends und können unverändert übernommen werden. Der einzige Teil, der angepasst werden muss, ist die Ausgabe des Maschinencodes. Üblicherweise wird der Code immer für eine komplette Funktion ausgegeben. In diesem Fall ist es jedoch erwünscht für jeden Basisblock getrennt den Assemblercode zu generieren. Dazu wird der Assemblerdrucker des Backends modifiziert.

Für jeden Basisblock wird dabei eine extra Datei angelegt, in die der generierte Assemblercode geschrieben wird. Damit sich der so generierte Assemblercode auch assemblieren lässt, wird noch eine Einsprungmarke benötigt. Am Ende eines Basisblocks stehen meist Sprunganweisungen, die auf die Sprungmarken der nachfolgenden Basisblöcke verweisen. Diese Sprungmarken müssen ebenfalls in der Datei enthalten sein, damit der generierte Code auch fehlerfrei assembliert werden kann. Am Ende des Prozesses existiert nun für jeden Basisblock eine Assemblerdarstellung.

Analyse des Assemblercodes

Der im vorigen Schritt erzeugte Assemblercode muss nun noch vom aiT-WCET-Analysewerkzeug untersucht werden, um seine WCET zu bestimmen. Zuerst wird dieser mit Hilfe des GNU Assemblers für die PowerPC Architektur assembliert, so dass die einzelnen Basisblöcke als Objektcode vorliegen. Der Objektcode wird dann vom ait-Analyzer untersucht. Dieser wird dazu im Stapelverarbeitungsmodus betrieben. Dazu werden Einstellungs- und Steuerungsdateien für jede Objektdatei generiert, die dann mit dem Objektcode des jeweiligen Basisblocks als Eingabe für die aiT-Analyse dienen. Die Ergebnisse des aiT-Analyzers finden sich nach Durchführung der Analyse in einer Ergebnisdatei, deren Auswertung die WCET des analysierten Basisblocks liefert. Sie wird dann im T-Graph in der dem Basisblock entsprechenden Kante vermerkt. Wurde dieser Schritt für alle Basisblöcke durchgeführt, dann ist die Zeitinformation im T-Graph vollständig.

3.2.8 Berechnung der WCET

Im letzten Schritt der Analyse wird aus den nun vollständigen T-Graphen die WCET der gewünschten Funktionen bestimmt. Dafür liegen nach Abarbeitung der in den vorigen Kapiteln beschriebenen Passes folgende Daten für jede Funktion vor:

- Abbildung des Kontrollflusses und der Ausführungszeiten der Basisblöcke in Form eines T-Graphen
- Restriktionen für die Anzahl der Schleifenausführungen
- Art der Funktionsaufrufe für jeden Basisblock
- Tiefe von rekursiven Funktionsaufrufen

Für einfache Funktionen, die keine weiteren Funktionen aufrufen, würde es genügen den entsprechenden T-Graphen in ein LP-Problem zu konvertieren und dieses dann zu lösen. Um jedoch Funktionsaufrufe innerhalb einer Funktion zu unterstützen, muss die Ausführungszeit eines jeden Basisblocks, in dem ein Funktionsaufruf stattfindet, um die WCET der aufgerufenen Funktion inkrementiert werden. Ist die aufgerufene Funktion zusätzlich noch rekursiv, dann muss zur Basisblockausführungszeit noch die Abarbeitungszeit für alle Funktionen des Aufrufzykluses addiert werden. Die Rekursionstiefe gibt dann an, wie oft der Aufrufzyklus durchlaufen wird. Für jeden Durchlauf addiert sich dann die Ausführungszeit des Zyklus zur Basisblockausführungszeit. Zur Bestimmung der WCET einer Funktion mit Funktionsaufrufen werden also, wie beschrieben, die einzelnen Funktionsaufrufe aufgelöst, um nun endgültige Gewichte für die einzelnen Kanten des T-Graphs zu bekommen. Dazu muss man die WCET aller aufgerufenen Funktionen berechnen. Wird eine Funktion mehrmals innerhalb der analysierten Funktion aufgerufen, dann wird ihre WCET lediglich ein einziges mal berechnet und dieser Wert bei den folgenden Aufrufen verwendet. Der Kontext des jeweiligen Prozeduraufrufs, also die Belegung der Eingabeparameter, fließt dabei nicht mit in die Berechnung der WCET mit ein.

Nachdem das für jeden Funktionsaufruf durchgeführt wurde, kann man den T-Graph in ein LP-Problem konvertieren, dieses dann lösen und erhält somit die WCET der Funktion (siehe Kapitel 2.1.5). Darüberhinaus gibt die Variablenbelegung an, wie oft jede der Kanten, und damit die dazugehörigen Basisblöcke, bei der zur WCET gehörigen Abarbeitung des Programmes ausgeführt werden.

3.3 Zusammenfassung

In diesem Kapitel wird der grundlegende Entwurf und die Implementierung eines WCET-Analysewerkzeugs für LLVM-Zwischencode erläutert. Zuerst wird dabei die Struktur der Analyse erläutert und aufgezeigt, wie sie sich innerhalb des LLVM-Frameworks einfügt. Dabei wird das Programm, welches als LLVM-Zwischencode vorliegt, in einen T-Graphen konvertiert und dieser Schritt für Schritt mit den notwendigen Informationen angereichert, damit dieser am Ende dieses Prozesses die zeitlichen Abläufe des Eingabeprogrammes umfassend beschreibt. Anschließend werden einige Teilbereiche des Entwurfs genauer beleuchtet, um die dabei getroffenen Entwurfsentscheidungen genauer darzustellen. Dies betrifft die Codegenerierung für die einzelnen Basisblöcke der LLVM-Zwischensprache sowie die Bereitstellung von zusätzlichen zeitlichen Informationen, die sich nicht direkt im Quellprogramm befinden. Auch die Behandlung von rekursiven Funktionen wird dabei näher beschrieben.

Nach der Beschreibung des Entwurfs wird die konkrete Implementierung beschrieben und dabei aufgezeigt wie sich die einzelnen Aufgaben innerhalb des LLVM-Frameworks realisieren lassen. Diese umfassen die einzelnen im Entwurf genannten Aufgaben wie die Erzeugung des T-Graphen und dessen Komplettierung mit den Informationen über Schleifen, Funktionsaufrufe und schließlich die Ausführungszeiten der einzelnen Basisblöcke.

4 Evaluation

Im nun folgenden Kapitel soll das Programm `llvmwcet`, dessen Implementierung in Kapitel 3 beschrieben wird, mit Hilfe einiger Beispiele getestet und damit das vorgestellte Konzept zur WCET-Analyse auf LLVM-Zwischencodeebene evaluiert werden. Als Referenz dient dabei das aiT-WCET-Analysewerkzeug, welches auch zur Bestimmung der Basisblockausführungszeiten verwendet wurde. Dadurch soll festgestellt werden, inwieweit der Ansatz, auf einer abstraktionell höheren Ebene die WCET zu bestimmen, die Ergebnisse im Vergleich zu einer Analyse auf Assemblerebene beeinträchtigt.

4.1 Bubblesort

Zuerst soll hier ein einfacher Bubblesort-Algorithmus (siehe [14, S.106–110]) evaluiert werden. Dieser wurde aufgrund seiner Struktur, bestehend aus zwei ineinander verschachtelten Schleifen gewählt, um den Umgang des WCET-Analysewerkzeuges mit solchen Schleifenkonstrukten zu veranschaulichen. Er wird auch in [8] und [13] verwendet.

In Abbildung 4.1 ist der verwendete C-Quellcode für den Bubblesort-Algorithmus angegeben. Man sieht, dass die äußere Schleife eine feste, im Quellcode vorgegebene Grenze aufweist. Somit kann das LLVM-Framework die maximale Anzahl der Schleifenausführungen selbst bestimmen und an dieser Stelle ist somit keine Annotation notwendig. Für die innere Schleife gilt dies jedoch nicht, da die Abbruchbedingung von der Laufvariablen der äußeren Schleife abhängt, so dass in Zeile 7 eine Annotation eingefügt wurde, um die maximale Anzahl der Schleifenausführungen vorzugeben.

Der Quellcode wurde mit drei verschiedenen Optimierungsstufen übersetzt: Mit `-O0`, was keinen Optimierungen entspricht, mit `-O1`, was grundlegende Optimierungen umfasst, und mit `-Os`, was das Kompilat auf Codegröße hin optimiert. Die Übersetzung wird dabei mit dem Werkzeug *llvm-gcc* durchgeführt, welches das Ergebnis als LLVM-Zwischencode erzeugt. Dieser

```
1  void bubbleSort(int a[]) {
2      int i;
3      int j;
4      int size = 10;
5      for(i = 0; i < (size-1); ++i) {
6          for (j = 0; j <= i; ++ j) {
7              setVariableLoopCount(size - 1, 9);
8              if (a[j] > a[j+1]) {
9                  swap(&a[j], &a[j+1]);
10             }
11         }
12     }
13     return;
14 }
```

Abbildung 4.1: Bubblesort in C-Code

kann dann direkt von `llvmwcet` analysiert werden. Zur Analyse mit dem aiT-WCET-Analysewerkzeug wird der Zwischencode noch vom LLVM-Werkzeug *llc* in PowerPC Assembler übersetzt und dann mit den *GNU binutils* für die PowerPC-Architektur assembliert und gebunden. Falls notwendig, wurde dem aiT noch eine Datei mit Annotationen für den Objektcode mit übergeben, um die Analyse durchführen zu können.

Bubblesort	**-O0**	**-O1**	**-Os**
llvmwcet	18607	8527	6304
aiT	17073	6528	4366
Δ	1534	1999	1938
Δ in %	9%	31%	44%

Tabelle 4.1: Ergebnisse der WCET-Berechnung

In Tabelle 4.1 sind die Ergebnisse der durchgeführten Messungen dargestellt. Alle berechneten Zeiten sind in Taktzyklen er CPU angegeben. Bei `-O0` ist `llvmwcet` um 1534 Zyklen schlechter als aiT. Das entspricht 9% und ist damit als guter Wert zu bezeichnen. Bei eingeschalteten Optimierungen vergrößert sich der Vorsprung des aiT-Werkzeuges jedoch beträchtlich auf 1999 Zyklen oder 31% (`-O1`) bzw. 1938 Zyklen oder 44% (`-Os`).

Diese Ergebnisse lassen sich jedoch bei Betrachtung des Algorithmus und der jeweilig verwendeten Analysemethode auf einfache Weise erklären. Bei dem ohne Optimierungen übersetzten Beispiel müssen für beide verwendeten Analysewerkzeuge die Grenzen für die Anzahl der Ausführungen der im Algorithmus auftretenden Schleifen annotiert werden, da weder `llvmwcet` mit

Hilfe der LLVM noch der aiT diese automatisch bestimmen können. Dies ist darin begründet, dass der bei der Übersetzung mit llvm-gcc generierte Code keine Variablen in Registern hält, sondern diese nach jeder Veränderung in den Speicher schreibt und bei jeder neuen Verwendung wieder aus diesem lädt. Dadurch wird die Erkennung der Schleifengrenze für die äußere Schleife durch die LLVM verhindert. Die innere muss bei Verwendung von `llvmwcet` in jedem Fall annotiert werden. Für den aiT muss nur für die Übersetzung ohne Optimierung annotiert werden. Sobald die Optimierung des Übersetzers aktiviert wird, kann er selbstständig für beide Schleifen bestimmen, wie oft diese maximal ausgeführt werden.

Durch die Art und Weise, wie der aiT diese Schleifengrenzenerkennung durchführt ist auch zu erklären, dass er, sobald Optimierungen zugeschaltet werden, ein signifikant besseres Ergebnis wie `llvmwcet` liefert. Wenn man sich den Bubblesort-Algorithmus genauer ansieht, dann fällt dabei auf, dass die innere Schleife nicht bei jeder Iteration der äußeren Schleife die maximale Anzahl von Iterationen, also neun mal ausgeführt wird. Dies ist jedoch das annotierte Maximum, welches aber nur in einem einzigen Fall, nämlich bei der letzten Iteration zum Tragen kommt. Der aiT-Analyzer betrachtet dabei jede Schleifenausführung einzeln. So wird erkannt, dass bei der ersten Iteration der äußeren Schleife die innere lediglich einmal ausgeführt wird, da diese den aktuellen Stand der Laufvariablen der äußeren Schleife als Grenze hat. Bei der zweiten Iteration der äußeren Schleife wird die innere dann zwei mal ausgeführt. Dies setzt sich durch alle Iterationen der äußeren Schleife hindurch fort. Bei der letzten wird schließlich die maximale Anzahl an Schleifendurchläufen für die innere Schleife erreicht. Durch diese Technik fließen in die WCET-Berechnung des aiT bedeutend weniger Iterationen der inneren Schleife mit ein als bei `llvmwcet`, welches für jede Iteration der äußeren Schleife die maximalen neun Iterationen der inneren Schleife berücksichtigt. Dadurch ist der prozentual vergrößerte Unterschied bei den Messungen mit `-O1` und `-Os` im Vergleich zur Messung mit `-O0` zu erklären.

Im Gegensatz dazu ist bei der Übersetzung ohne Optimierungen bei beiden Werkzeugen die Schleifendurchlaufzahl per Annotation direkt angegeben. Der dadurch geringere Unterschied von 9% ist durch die vom aiT verwendete Pipeline- und Cacheanalyse zu erklären. Während `llvmwcet` lediglich die einzelnen Basisblöcke der Funktion durch den aiT analysieren lässt, um deren Ausführungszeit zu ermitteln, übernimmt der aiT bei der Vergleichsmessung die komplette Analyse. Dies hat zur Folge, dass das Verhalten von Pipelines und Caches über die komplette Ausführung der Funktion modelliert werden kann. Geschieht diese Analyse jedoch für jeden Basisblock einzeln, dann wird der Effekt, den im Kontrollfluss vorrausgegangene Basisblöcke auf den inneren Zustand des Prozessors ausüben, unterschlagen und die Analyse beginnt für jeden Basisblock mit leeren Caches und Pipelines und den damit verbundenen negativen Folgen für die Laufzeitabschätzung. Sind die Basisblöcke kurz, so tritt dieser Effekt um so mehr zu Tage. Je länger die einzelnen Basisblöcke jedoch sind, desto besser und weniger pessimistisch

wird die Laufzeitabschätzung für die einzelnen Codesequenzen.

4.2 Steuerung für einen Quadrocopter

Der hier zur Evaluation verwendete Bubblesort-Algorithmus ist jedoch kein Beispiel, welches in der Praxis in eingebetteten Systemen, die kritische Aufgaben ausführen sollen, typischerweise vorkommt. Er wurde weniger aus Praxisrelevanz heraus gewählt, sondern um die Reaktion der beiden WCET-Analysewerkzeuge auf verschachtelte Schleifen zu untersuchen. Aus diesem Grund sollen die beiden Werkzeuge zusätzlich noch mit Programmcode aus praktischen Beispielen evaluiert werden.

Das erste Beispiel ist dabei der Steuerungsalgorithmus des I4Copter-Projekts[3]. Dieser umfasst im Kern den in Abbildung 4.2 dargestellten Code.

```
1  float control(float realphi, float realomega) {
2    om_dot = (1 / e->getInertia()) * (u - e->getEngineConst() * o_dot)
3           + C_D2 * (o - realomega);
4    o_mot += om_dot * C_T_ACT;
5    float o_dot = C_KJ * o_dot + C_D1 * (o - realomega);
6    o += o_dot * C_T_ACT;
7    phi += o * C_T_ACT;
8    u = C_K1 * o_dot + C_K2 * realomega + C_K3 * realphi;
9    return u;
10 }
```

Abbildung 4.2: Steuerungscode für den I4Copter

Dieser Algorithmus dient dazu, die Lage des Quadrocopters im Raum stabil zu halten. Dazu werden aus Messwerten der Lage und Beschleunigung des Quadrocopters die notwendigen Korrekturen mit obigem Code ermittelt. Bei Betrachtung des Quellcodes fällt auf, dass hier keinerlei Kontrollflussverzweigungen auftreten. Übersetzt wurde das Quellprogramm dabei einmal ohne (`-O0`) und einmal mit Optimierungen (`-Os`). Dabei ergeben sich die in Tabelle 4.2 dargestellen Ergebnisse.

Quadrocoptersteuerung	**-O0**	**-Os**
llvmwcet	637	189
ait	484	189

Tabelle 4.2: Ergebnisse für den Quadrocoptersteuerungsalgorithmus

Für den unoptimierten Fall ergibt sich wieder ein Vorteil für das ait-WCET-Analysewerkzeug, da der eigentlich sequenzielle Kontrollfluss des Programmes vom Frontend des verwendeten Compilers llvm-gcc nicht in sequenziellen Zwischencode übersetzt wurde. Auch die beiden Methodenaufrufe (`e->getInertia()` und `e->getEngineConst()`) in Zeilen 2 und 3 von Abbildung 4.2 tragen zu diesem Vorsprung bei, da der aiT sie im Kontext der einzelnen Aufrufe analysieren kann. Bei `llvmwcet` ist dies nicht der Fall. Das sorgt für ähnliche Auswirkungen wie die in Kapitel 4.1 beschriebene, aus dem Kontext herausgelöste Analyse der Basisblöcke: Die Pipeline- und Cacheanalyse muss im Fall von `llvmwcet` mit leeren Pipelines und Caches starten, was die von der Analyse gelieferte WCET negativ beeinflusst.

Schaltet man jedoch bei der Programmübersetzung Optimierungen zu (`-Os`), dann liefern beide Methoden dieselbe WCET. Durch die Optimierungen wird das komplette Programm in einen einzigen, großen Basisblock gepackt. Auch die beiden vorher noch vorhandenen Methodenaufrufe werden eingebettet. Es gibt dann nur noch einen einzigen Basisblock, und so ist das durch beide Werkzeuge erzielte Ergebnis deckungsgleich, was nicht verwunderlich ist, da in beiden Fällen derselbe Assemblercode durch den aiT analysiert wird.

Die beiden bisher gezeigten Beispiele zeigen zwei Extremfälle auf. Bei Bubblesort ist in praxisnahen Übersetzereinstellungen die durch `llvmwcet` berechnete WCET um 30-45% schlechter als die des aiT. Dies liegt, wie in den vorigen Abschnitten erläutert, an den Vorteilen bei der Analyse von komplexen Kontrollstrukturen durch den aiT. Als Gegenbeispiel dazu zeigt die Evaluation des Steuerungsalgorithmuses in der optimierten Version keinerlei Unterschied zwischen den beiden Analysewerkzeugen. Im Folgenden soll nun noch ein weiteres Beispiel vermessen werden, welches zum einen auch Praxisrelevanz besitzt, zum anderen jedoch nicht so einfach aufgebaut ist, wie der Steuerungsalgorithmus für den Quadrocopter.

4.3 Zustandsautomat für *„Hau den Lukas”*

Als letztes Evaluationsbeispiel kommt dabei ein Programm zum Einsatz, welches einen endlichen Zustandsautomaten modelliert. Dieser kommt als Teil der Lehrveranstaltung Echtzeitsysteme 2 zum Einsatz. Ein Metallstück kann durch das Magnetfeld mehrerer Spulen durch eine Plexiglasröhre bewegt werden. Durch den Zustandsautomaten wird beschrieben, an welcher Stelle und damit in welchem Zustand sich das Metallstück befindet und welche Spulen man ansteuern muss, um das Metallstück in eine bestimmte Richtung zu bewegen.

Der Programmcode in Abbildung 4.3 zeigt dabei den Zustandsübergang des primären Zustandsautomaten. Dieser führt, je nach Lage des Metallstückes in der Röhre zu weiteren Unterzuständen, sichtbar an den dazugehörigen Aufrufen in Zeilen 6, 14, 22, 30 und 38. Der eigentliche Zu-

```
1  ms_t FSM_Step() {
2    ms_t timeToNextStep = -1;
3    switch(state) {
4      case FSM_FETCH:
5        /* Fetch next command */
6        timeToNextStep = FSM_Fetch();
7        break;
8      case FSM_UP:
9        if (state_up == FSM_UP_END) {
10         /* Up-Fsm has finished, immediately get next command */
11         state = FSM_FETCH;
12         timeToNextStep = 0;
13       } else {
14         timeToNextStep = FSM_Up(); }
15       break;
16     case FSM_DOWN:
17       if (state_down == FSM_DOWN_END) {
18         /* Down-Fsm has finished, immediately get next command */
19         state = FSM_FETCH;
20         timeToNextStep = 0;
21       } else {
22         timeToNextStep = FSM_Down(); }
23       break;
24     case FSM_HOLD:
25       if (state_hold == FSM_HOLD_END) {
26         /* Hold-Fsm has finished, immediately get next command */
27         state = FSM_FETCH;
28         timeToNextStep = 0;
29       } else {
30         timeToNextStep = FSM_Hold(); }
31       break;
32     case FSM_RELEASE:
33       if (state_release == FSM_RELEASE_END) {
34         /* Release-Fsm has finished, immediately get next command */
35         state = FSM_FETCH;
36         timeToNextStep = 0;
37       } else {
38         timeToNextStep = FSM_Release(); }
39       break;
40     default:
41       break;
42   }
43   return timeToNextStep;
44 }
```

Abbildung 4.3: Implementierung des Zustandsüberganges für “Hau den Lukas”

standsübergang ist hier durch eine Switchanweisung dargestellt, welche abhängig vom gemessenen Eingabewert `state` die dazugehörigen Operationen ausführt. Die für die Behandlung der Unterzustände aufgerufenen Funktionen sind nach demselben Schema aufgebaut. Die hier abgebildete Funktion `FSM_Step` ist dabei der oberste Einstiegspunkt für den Übergang von einem Zustand in den nächsten. Berechnet man ihre WCET, so weiß man, wie lange ein Zustandsübergang maximal dauern kann. Die weiteren, von `FSM_Step` aufgerufenen Funktionen, welche die Übergänge in den Unterzuständen veranlassen, sind `FSM_Fetch`, `FSM_Up`, `FSM_Down`, `FSM_Hold` und `FSM_Release`. Die Übersetzung des Programmes erfolgt dabei wie in den vorigen Kapiteln beschrieben. Jedoch wird diesmal lediglich mit der Optimierungsstufe `-Os` gearbeitet, was eine maximale Praxisnähe gewährleistet.

FSM	FSM_Step	FSM_Fetch	FSM_Up	FSM_Down	FSM_Hold	FSM_Release
llvmwcet	389	237	307	290	284	276
aiT	328	199	255	235	236	217
Δ	61	38	52	55	48	59
Δ in %	19%	19%	20%	23%	20%	27%

Tabelle 4.3: Evaluationsergebnisse für die verschiedenen Teile der Zustandsmaschine

In Tabelle 4.3 sind die Ergebnisse für die einzelnen Funktionen der Zustandsmaschine dargestellt. `FSM_Step` ist dabei die Funktion, die aufgerufen wird, um einen Zustandsübergang auszulösen. Sie bedient sich dann, je nach Eingabewerten der anderen hier aufgeführten Funktionen, um den Zustandsübergang durchzuführen. Der Abstand der ermittelten WCET bewegt sich im Bereich von 20% bezogen auf die mit dem aiT ermittelte Ausführungszeit. Im Vergleich zum prozentualen Unterschied bei der Evaluation des Bubblesort-Algorithmuses, der bei Verwendung von `-Os` 44% beträgt, ist dies ein guter Wert. Der minimale Unterschied bei der Bubblesort Evaluation lag bei 9%, welcher ohne Verwendung von Optimierungen bei der Übersetzung zustande kam. Die hier erzielten Werte, die ca 20% schlechter sind als die Vergleichswerte, bewegen sich also zwischen diesen beiden Extremen. Im erzeugten Code tauchen bei diesem Beispiel keine komplizierteren Schleifenkonstrukte wie im Fall von Bubblesort auf, aber es existieren noch Funktionsaufrufe zum Behandeln der Unterzustände. Diese führen, wie auch schon beim Regelungsalgorithmus für den Quadrocopter in Kapitel 4.2, zu einer schlechteren Abschätzung der WCET durch `llvmwcet`.

4.4 Zusammenfassung

In diesem Kapitel wurde nun der vorgestellte Ansatz zur WCET-Analyse von Programmen in der LLVM-Zwischensprache mit Hilfe von einigen Beispielanwendungen untersucht und mit einer Analyse auf Assemblercodeebene verglichen. Dabei kamen drei unterschiedliche Evaluationsbeispiele zum Einsatz, von denen zwei aus realen, zeitkritischen Anwendungen stammen.

Als Ergebnis bleibt festzuhalten, dass bei realistischen Anwendungen aus dem Bereich der eingebetteten Systeme mittels der WCET-Analyse auf LLVM-Zwischencodeebene durchaus brauchbare Ergebnisse erzeugt werden können. Es ist dabei verschmerzbar, dass die direkte Analyse der Programmes durch den aiT durch dessen Fähigkeiten zur Pipeline- und Cacheanalyse, in den meisten Fällen zu einem einige Prozent besseren Ergebnis führt. Der Faktor hängt dabei entscheidend von der Komplexität des analysierten Kontrollflusses ab, so dass komplex verschachtelte Schleifenstrukturen mit vielen Funktionsaufrufen eine direkte Analyse des Objektcodes mehr bevorteilen als einfacher strukturierte Programme. Solche Strukturen sollten in zeitkritischem Code jedoch ohnehin möglichst vermieden werden, da diese allgemein schwerer analysierbar sind und somit ihr zeitliches Verhalten nur durch manuelle Vorgaben in Form von Annotationen vorhergesagt werden kann. Überwiegend werden also relativ einfach aufgebaute Programme verwendet, wie z.B. der hier aufgezeigte Zustandsautomat oder die im vorigen Kapitel verwendete Steuerung für den Quadrocopter. Die Analyseergebnisse für diese beiden Beispiele zeigen, dass sich die Laufzeitabschätzung durch `llvmwcet` durchaus im Bereich der Ergebnisse der direkten Analyse mit aiT befindet.

5 Ausblick und Erweiterungen

Im folgenden Kapitel sollen nun einige Erweiterungsmöglichkeiten für die in Kapitel 3 vorgestellte Implementierung aufgezeigt werden, die aus Aufwandsgründen dort nicht direkt berücksichtigt werden konnten.

5.1 Automatische Bestimmung von Schleifenschranken

In der jetzigen Implementierung kann nur für einfache Schleifen automatisch herausgefunden werden, wie oft diese maximal ausgeführt werden. Dabei wird auf eine im LLVM-Framework schon vorhandene Funktionalität zurückgegriffen, die jedoch nur eine sehr begrenzte Möglichkeiten zur Verfügung stellt. Ein Beispiel für eine Schleife, deren Anzahl an Iterationen durch die LLVM nicht vorhergesagt werden kann, ist die innere Schleife, des schon bei der Evaluation in Kapitel 4 verwendeten Bubblesort.

```
1  void bubbleSort(int a[]) {
2      int i;
3      int j;
4      int size = 10;
5      for(i = 0; i < (size-1); ++i) {
6          for (j = 0; j <= i; ++ j) {
7              if (a[j] > a[j+1]) {
8                  swap(&a[j], &a[j+1]);
9              }
10         }
11     }
12     return;
13 }
```

Abbildung 5.1: Bubblesort in C-Code

Dieser ist nochmals in Abbildung 5.1 abgebildet. Die Grenze für die äußere Schleife kann, wie in Kapitel 4.1 ausgeführt, von der LLVM erkannt werden. Bei der inneren Schleife funktioniert

dies jedoch nicht. Betrachtet man diese, so sieht man leicht, dass sie ebenfalls maximal so oft ausgeführt wird wie die äußere Schleife, jedoch nur bei der letzten Iteration der äußeren Schleife. Ziel soll es nun sein für solche Schleifenkonstrukte automatisch die Grenzwerte für die Anzahl der Ausführungen zu finden.

5.1.1 Wertebereichsabschrankung für Variablen

Die Grenze für die innere Schleife bildet die Variable `i`. Diese ist aber nicht mit einer Konstanten belegt, sondern ist, je nachdem in welcher Iteration sich die äußere Schleife befindet, variabel. Aus dem Code in Zeile 5 ist jedoch ersichtlich, dass die Variable `i` minimal 0 und maximal 9 ist. Man sieht also an dieser Stelle, dass die innere Schleife nie öfter als neun mal ausgeführt werden wird. Es wäre also sinnvoll für Variablen, die sich durch *Constant Propagation* und *Constant Folding* nicht auf einen konstanten Wert festlegen lassen, durch eine Analyse des Wertebereichs herauszufinden, welchen maximalen Wert sie jeweils annehmen können. Somit wäre es möglich in mehreren Fällen automatisch Grenzen für die Iterationsanzahl von Schleifen zu finden.

Diese Vorgehensweise bietet sich auch an, um den Wertebereich für Eingabeparameter bei Funktionsaufrufen einzuschränken. Bei der hier dargestellten Bubblesort-Implementierung ist die Größe des zu sortierenden Feldes fest vorgegeben. Der Funktionsprototyp für eine universellere Lösung würde in etwa so aussehen: `void bubbleSort(int a[], int size)`. Der Parameter `size` kann dann bei jedem Aufruf unterschiedlich sein und wird dann innerhalb der Prozedur als Variable verwendet. Über ihren Inhalt, abgesehen von den Einschränkungen, die durch den Variablentyp gegeben sind, kann jedoch keine genauere Angabe gemacht werden. In der bisherigen Implementierung müssen dadurch beide Schleifen mittels Annotationen mit Schranken versehen werden. Diese sind dann unabhängig vom Aufrufkontext und spiegeln die maximale Iterationszahl wieder, die für alle Aufrufe innerhalb des kompletten analysierten Programmes gelten müssen. Sind die Wertebereiche der Eingabevariablen jedoch hinreichend abschrankbar, so kann man für jeden Aufruf der Funktion individuell feststellen, wieviele Iterationen maximal möglich sind. Damit ist eine bessere WCET-Abschätzung möglich als wenn bei jedem Aufruf das Maximum über alle Aufrufe hinweg verwendet wird. Darüberhinaus kann man dadurch potentiell die Anzahl der notwendigen Annotationen reduzieren.

5.1.2 Symbolische Ausführung

Für die innere Schleife kann mit diesem Verfahren die maximale Iterationsanzahl von neun bestimmt werden, da der Wertebereich von `i` eingeschränkt werden kann. Jedoch wird diese

Anzahl von Iterationen lediglich bei der letzten Ausführung der äußeren Schleife erreicht. Eine weitere Verbesserung erreicht man dadurch, dass man die Schleifen symbolisch ausführt. So bekommt man für jede Iteration der äußeren Schleife einen individuellen Wert für `i`, der dann als Abbruchbedingung für die innere Schleife dienen kann. Die Anzahl der Iterationen der inneren Schleife wird hier also für jeden Aufruf dieser neu bestimmt. Dieses Vorgehen ist analog zu dem Verfahren, welches der aiT verwendet, um die Anzahl der Iterationen für Schleifen zu bestimmen (siehe Kapitel 4.1). Es handelt sich hier um Informationen, die auch auf der höheren Abstraktionsebene der LLVM-Zwischensprache vorhanden sind, so dass sich diese Berechnungen auch auf dieser Ebene ausführen lassen. Allerdings ist die Implementierung dieses Verfahrens aufwendig, weshalb es hier keine Beachtung fand.

5.2 Unterstützung anderer Compiler/Zielarchitekturen

Ein Nachteil des in dieser Arbeit behandelten Ansatzes zur WCET-Analyse ist, dass durch die Verwendung des LLVM-Frameworks die Anzahl der Zielarchitekturen auf die durch LLVM unterstützten beschränkt ist. Bei der normalen Verwendung des LLVM-Übersetzers gibt es die Möglichkeit anstatt direkt Assemblercode generieren zu lassen ein generisches Backend zu nutzen, welches stattdessen C-Code generiert. Dieser kann dann mit Hilfe eines jeden beliebigen C-Übersetzers in Code für die gewünschte Architektur übersetzt werden. Die gewählte Form der Assemblergenerierung für die einzelnen Basisblöcke des LLVM-Zwischencodes ist jedoch nicht einfach auf das C-Backend übertragbar. Bei Assemblercode verhält es sich so, dass einzelne, aus dem Kontext des Gesamtprogramms herausgenommene Fragmente immer noch ein syntaktisch korrektes Assemblerprogramm darstellen. Dadurch ist es möglich die einzelnen Assemblerfragmente durch den aiT analysieren zu lassen. Bei C-Code ist das jedoch nicht der Fall. Der einem Basisblock in der LLVM-Zwischensprache entsprechende C-Code ist für sich alleine genommen noch kein korrektes C-Programm. Ein solches wird jedoch benötigt, da der generierte C-Code nochmals mit einem Compiler übersetzt werden muss, um schlussendlich den gewünschten Assemblercode für die eigentliche Zielarchitektur zu erhalten.

Eine basisblockweise Codegenerierung mit Hilfe des C-Backends wäre jedoch mittels des in Kapitel 3.1.2 vorgestellten Ansatzes der Funktionsgenerierung aus Basisblöcken möglich. Da hier für jeden Basisblock schon auf LLVM-Ebene korrekte Funktionen generiert werden, ist es problemlos möglich diese über den Umweg des C-Backends schließlich in die Assemblersprache der eigentlichen Zielplattform zu übersetzen. Dabei muss man jedoch mit den schon erörterten Nachteilen dieser Methode bezüglich der Qualität der berechneten Ausführungszeiten leben. Zusätzlich bleibt noch anzumerken, dass die Qualität des vom C-Backend generierten Codes ebenfalls zu wünschen übrig lässt. Als Beispiel soll wiederum der in Kapitel 3.1.2

als Beispiel verwendete GGT-Algorithmus dienen. Dieser wird übersetzt für einen TriCore Mikrocontroller[7], für den kein natives LLVM-Backend existiert, und analysiert mit der zu diesem Prozessor passenenden Version des aiT. Bei direkter Übersetzung mittels der GCC ergibt sich für den generierten Code eine WCET von 68 Taktzyklen. Nimmt man dagegen den Umweg über das C-Backend der LLVM und übersetzt dann dessen Ausgabe wiederum mit der GCC, so errechnet der aiT eine WCET von 157 Taktzyklen. Daran sieht man, dass bei Verwendung des C-Backends potentiell weitere Einbußen der Analysequalität in Kauf genommen werden müssen.

5.3 Granularität der von aiT analysierten Einheiten

In der hier vorgestellten Implementierung der Analyse werden die Assemblerrepräsentationen der einzelnen Basisblöcke durch den aiT analysiert. Dies führt vor allem bei kurzen Basisblöcken dazu, dass die Ausführungszeiten der Basisblöcke als zu lange eingeschätzt werden. Ein erster Schritt um diese Situation zu verbessern, ist es, sequenzielle Basisblöcke in einem Stück analysieren zu lassen. Damit wird der negative Einfluss der bei jeder einzelnen Analyse neu gestarteten Pipeline- und Cachevorhersage minimiert. Verwendet man jedoch Optimierungen bei der Erzeugung des LLVM-Zwischencodes, so wird der Übersetzter schon bestrebt sein, die Anzahl von rein sequenziellen Basisblöcken gering zu halten, was den potentiellen Gewinn durch diese Maßnahme ziemlich gering erscheinen lässt.

Dehnt man die Größe der Analyseeinheiten auf baumartige Kontrollflusstrukturen aus, so ist es möglich einen noch größeren Teil des Programmes auf einmal zu analysieren, was die Ergebnisse weiter verbessern dürfte. Durch die Beschränkung auf baumartige Strukturen vermeidet man es, dass der aiT Schleifen analysieren muss. Für diese müssten unter Umständen Annotationen generiert werden. Dies ist jedoch nicht sehr einfach, da man dazu die Informationen, die auf LLVM-Zwischencodeebene vorliegen auf die Assemblerebene transferieren müsste. Diesen Prozess kann man aber nur schwer automatisieren. Ein weiteres Problem ist, dass wenn man komplette Schleifen außerhalb des ursprünglichen Kontextes analysieren lässt, dann funktioniert die eigentlich gute Vorhersage der Iterationsanzahl des aiT nicht mehr besonders gut, da dann die Belegung der Variablen eventuell nicht mehr bekannt ist, da diese vor der eigentlichen Schleife festgelegt wird.

Allgemein wird es also je nach Umfang der zu analysierenden Einheiten immer schwerer diese automatisiert vom aiT analysieren zu lassen.

5.4 Verwandte Arbeiten

Im Folgenden sollen nun noch einige, mit dem in dieser Arbeit behandelten Thema verwandte Ansätze zur Bestimmung der WCET vorgestellt werden.

5.4.1 WCET Analyse von Java Bytecode

In [8] wird eine Möglichkeit vorgestellt auf möglichst portable Weise Java Bytecode auf seine WCET hin zu untersuchen. Der Java Bytecode ist ähnlich die wie LLVM Zwischensprache von der jeweiligen Zielplattform unabhängig. Deshalb wird dort versucht die Zeitinformationen auf möglichst generische Weise zu beschreiben, um die WCET für einzelne Zielplattformen daraus einfach bestimmen zu können. Dabei wird aus dem Bytecode der Kontrollflussgraph extrahiert und durch Datenflussanalyse oder mit Hilfe von Annotationen die maximale Iterationstiefe von Schleifen bestimmt. Die geschieht auf eine vom jeweiligen Sprachfrontend unabhängige Weise, so dass nicht nur Bytecode, der aus Java Quellcode generiert wurde, analysiert werden kann. Dabei wird ein abstraktes Modell der jeweilig eingesetzten JVM [1] benutzt, um aus diesen Zeitinformationen, die sich auf einer konzeptionell relativ hohen Ebene befinden, die WCET des betrachteten Programmes zu bestimmten. Für die Evaluation wird ein JVM-Modell für einen PowerPC Prozessor betrachtet, das die einzelnen Bytecodebefehle unabhängig voneinander in Assembler übersetzt. Es kommen also keine Optimierungen zum Einsatz, die sich über mehr als einen Bytecodebefehl erstrecken, sondern jeder Befehl wird in eine definierte Folge von Assemblerinstruktionen überführt. Diese Übersetzung wird dabei von einem *Ahead of Time* (AOT) Compiler übernommen. Eine *Just in Time* Übersetzung im Kontext der WCET Bestimmung ist nicht sinnvoll, da sich dann zur Ausführungszeit des eigentlichen Codes noch die Dauer der Übersetzung addiert.

Darüberhinaus wird eine Methode vorgestellt, die es ermöglicht bei der Berechnung der Basisblockausführungszeiten die Effekte von Pipelines, wie sie in vielen modernen Prozessoren anzutreffen sind, mit in die Analyse einzubeziehen. Dabei wird zunächst der für eine Java Bytecode Instruktion generierte Assemblercode betrachtet und dann dessen Ausführung auf einem Modell einer Pipeline durch eine Tabelle beschrieben, in der die Belegung der einzelnen Stufen durch die jeweiligen Instruktionen gegen die Zeit aufgetragen ist. Durch Konkatenation dieser Tabellen erhält man eine verbesserte Abschätzung für die Nacheinanderausführung von verschiedenen Java Bytecodeinstruktionen, die Effekte durch eine Ausführung in einer Pipeline mit beinhaltet.

[1] Java Virtual Machine

Zur Vereinfachung der Analyse werden nur Paare von Bytecodeinstruktionen betrachtet. Dabei kann nun für jedes Paar von Bytecodeinstruktionen angegeben werden wie stark die Ausführung durch die Pipeline des Prozessors beschleunigt wird. Mit Hilfe dieser Information ist es jetzt möglich bei der Bestimmung der Basisblockausführungszeiten die Effekte durch Prozessorpipelines miteinzubeziehen.

Diese Arbeit zeichnet sich dadurch aus, dass er, ähnlich wie der hier vorgestellte Ansatz eine WCET-Berechnung auf einer höheren Abstraktionseben ausführt. Dabei wird allerdings nicht auf eine bestehende Compilerinfrastruktur aufgebaut, sondern durch einfaches Ersetzen der Java Bytecodeinstruktionen durch ihnen entsprechende Assemblersequenzen der Zielarchitektur eine Übersetzung durchgeführt. Dadurch ist für jede Java Bytecodeinstruktion bekannt, wie lange ihre Ausführung maximal dauern wird, da die einzelnen, kurzen Sequenzen einfach analysiert werden können. Die Ausführungszeiten für die einzelnen Basisblöcke können dann einfach durch Summenbildung bestimmt werden. Zur weiteren Verbesserung der Analyse wir dann der Effekt von Prozessorpipelines auf die Ausführungszeit des für ein Paar von Java Bytecodeinstruktionen generierten Assemblercodes bestimmt und in die Berechnung der WCET für Basisblöcke miteinbezogen.

5.4.2 Entwurf eines WCET-gewahren C-Übersetzers

In [11] wird ein Ansatz vorgestellt, eine WCET-Analyse direkt in den Übersetzer einzubinden und damit dann Optimierungen zu ermöglichen, die der Minimierung der WCET dienen. Dabei wird auf einen vorhandenen Compiler aufgesetzt, der Code für den Infineon TriCore [7] erzeugt. Die Analyse setzt dabei auf der maschinennahen Zwischensprache des Compilers (LLIR) auf, die das Eingabeprogramm auf Assemblerebene beschreibt. Die eigentliche Analyse wird mit dem auch in dieser Arbeit verwendeten aiT durchgeführt. Dabei kommt jedoch die Version für den TriCore Mikroprozessor zum Einsatz. Die LLIR wird dabei direkt in die interne Programmrepräsentation des aiT (CRL2) konvertiert und diese dem Analysewerkzeug dann direkt zur Verfügung gestellt. Die durch den aiT berechneten WCET-Informationen können dann in einem zur Abbildung von LLIR nach CRL2 inversen Schritt wieder dem Compiler zur Verfügung gestellt werden. Dies umfasst z.B. die WCET für einzelne Basisblöcke oder ganze Funktionen. Aber auch während der Analyse von aiT erzeugte Informationen wie die Wertebereiche von Registern oder die Zustände der Pipeline und des Caches können mit diesem Ansatz dem Compiler bekannt gemacht werden und stehen somit möglichen Optimierungen der WCET innerhalb des Übersetzers zur Verfügung.

Verglichen mit dem in dieser Arbeit vorgestelltem Ansatz wird hier die Analyse auf einer niedrigeren Abstraktionsebene durchgeführt. Die LLIR ist im Prinzip eine genaue Entsprechung der

Assemblersprache für den TriCoreprozessor. Innerhalb des LLVM-Frameworks entspricht dies der Maschinenrepräsentation, die im Backend aus der plattformunabhängigen LLVM-Zwischensprache generiert wird. Der Fokus liegt dabei darauf Optimierungen zu ermöglichen, um die WCET zu verbessern.

5.4.3 Die Programmiersprache WCETC

In den bisher vorgestellten Arbeiten und auch in der in dieser Arbeit realisierten Lösung werden Informationen, die für die WCET-Analyse benötigt werden, aber nicht aus der Quellsprache extrahiert werden können, dem Analysewerkzeug mit Hilfe von Annotationen bereitgestellt. In [13] wird im Gegensatz dazu ein Ansatz vorgestellt, um diese Informationen direkt über Erweiterung der Quellsprache direkt im Programmcode zu verankern. Dabei wurde der Sprachumfang der Programmiersprache C um einige Schlüsselwörter erweitert, die es ermöglichen Informationen über den zeitlichen Ablauf des Programmes anzugeben. So wurde die Definition von do, while und for-Schleifen angepasst, damit man dort die maximale Iterationsanzahl angeben kann. Darüberhinaus kann man mit Hilfe von Restriktionen die Anzahl der Ausführungen für bestimmte Pfade direkt angeben oder von der Ausführung von anderen Pfaden abhängig machen. Diese Möglichkeiten erinnern an die in Kapitel 2.1.4 vorgestellten Flussrestriktionen. Auf diese Weise kann man jedoch direkt im Programmquellcode solche Restriktionen verankern und hat dadurch mehrere Möglichkeiten als mit einfachen Ausführungsgrenzen für Schleifen.

Im Vergleich zu ANSI-C existieren noch einige weitere Einschränkungen, um eine Zeitanalyse zu ermöglichen. Der Gebrauch von Funktionszeigern wird untersagt, damit sichergestellt wird, dass bei Funktionsaufrufen schon zur Übersetzungszeit bekannt ist, welches die aufgerufene Funktion ist. Auch rekursive Funktionen sind verboten. Die Benutzung der den normalen Kontrollfluss verändernden Instruktionen `goto` und `setjmp()/longjmp()` ist ebenfalls nicht erlaubt.

Der Fokus dieser Arbeit liegt dabei auf der Gestaltung der Schnittstelle zur Quellsprache. Diese soll eine möglichst gute Beschreibung des zeitlichen Verhaltens ermöglichen, damit die Qualität der WCET-Analyse verbessert wird.

5.5 Zusammenfassung

In diesem Kapitel werden verschiedene Erweiterungen für die Implementierung diskutiert und kurz skizziert. Im Anschluss daran soll noch ein kurzer Einblick in einige verwandte Arbeiten gegeben werden, die teilweise ähnliche Ziele verfolgen die diese.

Erweiterungen der WCET-Analyse sind an verschieden Stellen der Implementierung möglich. So bietet es sich an auf der Abstraktionsebene der LLVM-Zwischensprache zu versuchen mehr Informationen über den zeitlichen Ablauf des Programmes zu gewinnen. Dies betrifft vor allem Abschrankungen der Iterationszahl von Schleifen. Durch Abschrankung des Wertebereichs für die in der Schleifenabbruchbedingung verwendeten Variablen und durch symbolische Ausführung des dazugehörigen LLVM-Zwischencodes lassen sich mehr Schranken für die Ausführung von Schleifen ermitteln als dies mit der schon in LLVM vorhanden Methode möglich ist. Auch das Einbeziehen von Kontexten z.B. bei Funktionsaufrufen kann die Analyse verbessern. Einen weiteren Ort für mögliche Erweiterungen stellt das verwendete Backend dar. Hier ist es interessant die Anzahl an unterstützten Architekturen auch über die in LLVM direkt vorhandenen hinaus zu vergrößern. Die Ansteuerung des externen Analysewerkzeuges ist eine Möglichkeit die Güte der Analyse zu verbessern, indem die einzelnen analysierten Objektcodestücke so groß wie möglich gemacht werden, um die sehr genaue Analyse durch den aiT bezüglich Prozessorpipelines und Cache besser auszunutzen.

Ein zu dieser Arbeit ähnlicher Ansatz ist die WCET-Analyse von Java Bytecode. Dieser ist, wie die LLVM-Zwischensprache, eine maschinenunabhängige Form der Programmdarstellung. Dabei wird jedoch kein üblicher Compiler verwendet, sondern die einzelnen Java Bytecodeinstruktionen werden in genau definierte Assemblersequenzen der Zielarchitektur übersetzt. Zur Verbesserung der Analyse werden noch die positiven Effekte von Prozessorpipelines auf Ausführung von Bytecodeinstruktionspaaren berechnet und verbessern so das Ergebnis.

Der WCET-gewahre C-Übersetzer mit einem Fokus auf Optimierungen, welche die WCET verbessern sollen, setzt dagegen auf einer etwas niedrigeren Ebene an, nämlich der maschinenspezifischen Zwischensprache des Compilers. Diese wird im Laufe der Analyse in die Programmrepräsentation des verwendeten aiT-Analyzers konvertiert und die damit erzielten Aussagen über das Zeitverhalten des Programmes wieder in den Übersetzer zurückübertragen. So stehen die zeitlichen Informationen dort den Optimierungen zur Verfügung.

Zuletzt wird noch eine Spracherweiterung für C betrachtet, mit der es möglich ist, die für die WCET-Analyse notwendigen Informationen direkt im Quellcode mit Unterstützung der Sprachsyntax zu spezifizieren. Damit lassen sich auch direkt Flussrestriktionen festlegen.

6 Resümee

Im Laufe dieser Arbeit wird eine Methode vorgestellt, um auf Basis einer abstrakten Programmbeschreibung, in diesem Fall die LLVM-Zwischensprache, die WCET für darin verfasste Programme zu bestimmen. Zuerst werden die Grundlagen der WCET-Bestimmung mit Hilfe von Flüssen auf T-Graphen erläutert und im Anschluss daran der grundsätzliche Aufbau der LLVM und dessen Implikation für die WCET-Analyse beschrieben.

Nach der Erläuterung der Grundlagen folgt nun ein Überblick über den Entwurf der WCET-Analyse. Dabei wird beschrieben, wie sich die Analyse grundlegend in den Aufbau des LLVM-Frameworks einfügen lässt. Im Anschluss daran werden einige zentrale Probleme, die sich bei einer Analyse auf einer höheren Abstraktionsebene ergeben, näher erläutert und die getroffenen Entwurfsentscheidungen begründet. Hierauf folgt dann eine Beschreibung der Implementierung. Es wird also dargelegt, wie die einzelnen Aufgaben innerhalb des Entwurfs konkret realisiert werden. Dabei wird auch aufgezeigt, wie sich die Analyse in die Struktur des LLVM-Frameworks einweben lässt und dort schon vorhandene Funktionen verwenden kann.

Im darauffolgenden Kapitel wird nun die Implementierung mit Hilfe einiger Beispielapplikationen evaluiert. Dabei werden die dafür berechneten maximalen Ausführungszeiten mit denen durch ein anderes Werkzeug ermittelten verglichen, welches die Analyse direkt auf dem erzeugten Objektcode durchführt. Es wird dann untersucht, in wieweit der Ansatz auf LLVM-Zwischencodeebene die Analyse negativ beeinflusst und wie stark dies in verschiedenen Szenarien zutage tritt. Am Ende schließt sich noch ein Ausblick an, in dem Erweiterungen und Verbesserungsmöglichkeiten für den Ansatz zur WCET-Bestimmung diskutiert werden und einige, ähnliche Projekte aus der Forschung kurz vorgestellt werden.

Mit der in dieser Arbeit vorgestellten Lösung wird es ermöglicht aus der abstrakten Programmbeschreibung der LLVM die WCET des Programms für eine bestimmte Zielplattform zu bestimmen. Die dabei berechneten Ausführungszeiten sind, wie die Evaluation in Kapitel 4 gezeigt hat, in vielen Fällen in der Größenordnung einer Low-Level Analyse angesiedelt. Je nach Struktur des analysierten Programmes können sich jedoch auch größere Abweichungen ergeben. In praxisrelevanten Szenarien ergeben sich insgesamt recht gute Ergebnisse. Daraus kann geschlossen

werden, dass eine Analyse, die nicht auf der untersten Ebene des Objektcodes durchgeführt wird, durchaus praxistaugliche Ergebnisse erzielen kann.

Darüberhinaus hat der hier vorgestellte Ansatz auch weitere Vorteile: So ist es durch Verwendung der LLVM-Plattform mit relativ geringem Aufwand möglich die Zielarchitektur zu wechseln. Dazu muss lediglich die Assemblerausgabe für die Zielarchitektur angepasst werden, so dass diese Code für die einzelnen Basisblöcke erzeugt. Einen weiteren Vorteil hat man duch die Verwendung von Annotationen direkt im Quelltext des Programmes. Dies ist zwar bei Benutzung des aiT ebenfalls möglich, jedoch beschränkt sich dessen Unterstützung auf die C Programmiersprache. Der Annotationsmechanismus, der in dieser Arbeit verwendet wird, kann jedoch auf beliebige Quellsprachen ausgeweitet werden.

Die angestrebte Verwendung der Analyse im Rahmen eines RTSC (vgl. Kapitel 1.1) ist also durchaus vertretbar. Die erreichten Analyseergebnisse sind für diesen Zweck ausreichend gut und die Flexibilität aufgrund der Verwendung des LLVM-Frameworks ermöglicht es die Analyse auf weitere Architekturen zu portieren.

A Anhang

A.1 Wichtige LLVM Instruktionen

Terminatorinstruktionen	
ret	Kehrt aus einer Funktion zurück. Rückgabewert ist der erste Parameter
br	Verzweigt den Kontrollfluss zwischen zwei Basisblöcken
switch	Verzweigt den Kontrollfulss zwischen zwei und mehr Basisblöcken
invoke	Aufruf einer Funktion mit Möglichkeit zur Ausnahmebehandlung
unwind	Stößt Ausnahmebehandlung an
Logische und arithmetische Instruktionen	
add, sub, mul	Addition, Subtraktion und Multiplikation für Variablen gleichen Typs
udiv, sdiv, fdiv	Division für ganzzahlige vorzeichenlose, vorzeichenbehaftete und für Fliesspunktvariablen
urem, srem, frem	Berechnung des Divisionsrestes für ganzzahlige vorzeichenlose, vorzeichenbehaftete und für Fliesspunktvariablen
shl	Linksshift
lshr	Logischer Rechtsshift
ashr	Arithmetischer Rechtsshift
and, or, xor	Logisches Und, Oder bzw exklusives Oder
Speicherzugriffsinstruktionen	
malloc	Speicherallokation auf dem Heap
free	Speicherfreigabe auf dem Heap
alloca	Speicherallokation auf dem Strack
load	Lesender Zugriff auf Speicher
store	Schreibender Zugriff auf Speicher
getelementptr	Zusammengefasste typsichere Indexierungs- und Dereferenzierungsoperation
Andere Instruktionen	
icmp, fcmp	Vergleich von Integer- oder Gleitkommawerten
phi	Auswahl von Werten abhängig vom vorher ausgeführten Basisblock
select	Auswahl eines Wertes abhängig von einer vorhergehenden Vergleichsoperation
call	Aufruf von Funktionen ohne Ausnahmebehandlung

Tabelle A.1: Beschreibung einiger wichtigen LLVM Instruktionen

A.2 Iterativer GGT Algorithmus als LLVM-Zwischencode

```
1  define i32 @gcd(i32 %argc, i8** %argv) nounwind {
2  entry:
3    %tmp2 = mul i32 %argc, 10
4    %tmp4 = sdiv i32 %tmp2, 2
5    %tmp2.off = add i32 %tmp2, 1
6    %tmp1736 = icmp ult i32 %tmp2.off, 3
7    br i1 %tmp1736, label %bb20, label %bb.outer
8
9  bb.outer:
10   %a.025.0.ph = phi i32 [ %tmp2, %entry ], [ %tmp11, %cond_true ]
11   %b.030.0.ph = phi i32 [ %tmp4, %entry ], [ %b.030.0, %cond_true ]
12   br label %bb
13
14 bb:
15   %indvar = phi i32 [ 0, %bb.outer ], [ %indvar.next, %bb15 ]
16   %tmp. = sub i32 0, %a.025.0.ph
17   %tmp.42 = mul i32 %indvar, %tmp.
18   %b.030.0 = add i32 %tmp.42, %b.030.0.ph
19   %tmp7 = icmp sgt i32 %a.025.0.ph, %b.030.0
20   br i1 %tmp7, label %cond_true, label %bb15
21
22 cond_true:
23   %tmp11 = sub i32 %a.025.0.ph, %b.030.0
24   %tmp1738 = icmp eq i32 %b.030.0, 0
25   br i1 %tmp1738, label %bb20, label %bb.outer
26
27 bb15:
28   %tmp17 = icmp eq i32 %b.030.0, %a.025.0.ph
29   %indvar.next = add i32 %indvar, 1
30   br i1 %tmp17, label %bb20, label %bb
31
32 bb20:
33   %a.025.1 = phi i32 [ %tmp2, %entry ], [ %a.025.0.ph, %bb15 ],
34                       [ %tmp11, %cond_true ]
35   ret i32 %a.025.1
36 }
```

Literaturverzeichnis

[1] *aiT-WCET-Analysewerkzeug*. http://www.absint.com/ait/index_de.htm.

[2] *Elektronischer "Hau den Lukas"*. http://www4.informatik.uni-erlangen.de/ /Lehre/SS07/V_EZS2/Uebung/hau_den_lukas.shtml.

[3] *The I4Copter Project*. http://www4.informatik.uni-erlangen.de/ ulbrich/I4Copter/.

[4] *Low Level Virtual Machine*. http://www.llvm.org/.

[5] *lp_solve Library Reference Guide*. http://lpsolve.sourceforge.net/5.5/.

[6] *PowerPC MPC565 Produktübersicht*. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=MPC565&nodeId=016246PCbf8648.

[7] *TriCore Produktüberblick*. http://www.infineon.com/cms/en/product/channel.html?channel=ff80808112ab681d0112ab6b73d40837.

[8] BATE, IAIN, GUILLEM BERNAT, GREG MURPHY und PETER PUSCHNER: *Low-Level Analysis of a Portable Java Byte Code WCET Analysis Framework*. In: *Proc. 7th International Conference on Real-Time Computing Systems and Applications*, Seiten 39–48, Dec. 2000.

[9] CYTRON, RON, JEANNE FERRANTE, BARRY K. ROSEN, MARK N. WEGMAN und F. KENNETH ZADECK: *Efficiently Computing Static Single Assignment Form and the Control Dependence Graph*. ACM Transactions on Programming Languages and Systems, 13(4):451–490, Oct 1991.

[10] DANEL, MICHAEL: *WCET-Analyse für ein Java-Echtzeitbetriebssystem*. Doktorarbeit, Friedrich-Alexander Universität Erlangen-Nürnberg, 2005.

[11] FALK, H., P. LOKUCIEJEWSKI und H. THEILING: *Design of a WCET-Aware C Compiler*. In: *ESTMED '06: Proceedings of the 2006 IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, Seiten 121–126, Washington, DC, USA, 2006. IEEE Computer Society.

[12] ILJA BRONSTEIN, KONSTANTIN SEMENDJAJEW, GERHARD MUSIOL UND HEINER MÜHLIG: *Taschenbuch der Mathematik*. Harri Deutsch Verlag, 2000.

[13] KIRNER, RAIMUND: *The Programming Language wcetC*. Research Report 2/2002, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2002.

[14] KNUTH, DONALD: *The Art of Computer Programming*, Band 3: Sorting and Searching. Addison-Wesley, 1997.

[15] MACOS, DRAGAN und FRANK MUELLER: *Integrating Gnat/Gcc into a Timing Analysis Environment*. In: *In 10th EUROMICRO Workshop on Real Time Systems*, Seiten 15–18, 1998.

[16] NEMHAUSER, G.L., A.H.G. RINNOOY KAN und M.J. TODD: *Integer and Combinatorial Optimization*. John Wiley & Sons Ltd., 1988.

[17] PUSCHNER, PETER: *Zeitanalyse von Echtzeitprogrammen*. Doktorarbeit, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 1993.

[18] SCHELER, FABIAN, MARTIN MITZLAFF, WOLFGANG SCHRÖDER-PREIKSCHAT und HORST SCHIRMEIER: *Towards a Real-Time Systems Compiler*. Proceedings of the Fifth International Workshop on Intelligent Solutions in Embedded Systems (WISES 07) (Fifth International Workshop on Intelligent Solutions in Embedded Systems (WISES 07) Leganes (Madrid) 21./22.06.2007), Seiten 62–75.

[19] SCHELER, FABIAN und WOLFGANG SCHRÖDER-PREIKSCHAT: *Time-Triggered vs. Event-Triggered: A matter of configuration?* MMB Workshop Proceedings (GI/ITG Workshop on Non-Functional Properties of Embedded Systems Nuremberg 27.03. - 29.03.2006), Seiten 107–112.