Institut für Softwaretechnologie

Abteilung Programmiersprachen und Übersetzerbau

Universität Stuttgart
Universitätsstraße 38
D - 70569 Stuttgart

Studienarbeit Nr. 2108

Schwierigkeiten für statische Programmanalysen in der Praxis

Christoph Stach

Studiengang:	Informatik
Prüfer:	Prof. Dr. E. Plödereder
Betreuer:	Herr S. Staiger
begonnen am:	04.06.2007
beendet am:	04.12.2007
CR-Klassifikation:	D.2.8, D.3.3, D.3.4

Inhaltsverzeichnis

1 Einleitung **9**

2 Das Bauhaus Projekt **11**
2.1 Hintergrund 11
2.2 Die IML 11
2.3 Verwendete Tools aus dem Bauhaus Projekt 12

3 Aufgabenstellung **15**
3.1 Funktionszeigeranalyse 15
3.2 (Kontroll-) Strukturen und Aufrufbeziehungen 15
3.3 Interprozedurale Analyse 16
3.4 Zeiger- und Datenflussanalyse 16
3.4.1 Zeiger in Conditionals 16
3.4.2 Zeiger auf parallelen Strukturen 16

4 Testprogramme **19**
4.1 Anpassungen der Makefiles 25
4.2 Größe der Programme 26

5 Funktionszeiger-Analysen **31**
5.1 Bearbeitung 31
5.2 Analyseergebnisse 34
5.3 Auswertung 45

6 (Kontroll-) Strukturen und Aufrufbeziehungen **49**
6.1 Bearbeitung 49
6.2 Analyseergebnisse 53
6.3 Auswertung 57

7 Interprozedurale Analysen **61**
7.1 Bearbeitung 61
7.2 Analyseergebnisse 65
7.3 Auswertung 76

8 Zeiger- und Datenflussanalysen **79**
8.1 Zeiger in Conditionals 79
8.1.1 Bearbeitung 79

8.1.2 Analyseergebnisse 81
8.1.3 Auswertung 84
8.2 Zeiger auf parallele Strukturen 86
8.2.1 Bearbeitung 86
8.2.2 Analyseergebnisse 88
8.2.3 Auswertung 90

9 Abschlussbetrachtung **93**

Anhang **97**
Testrepository 97

Literaturverzeichnis **101**

Erklärung **103**

Abbildungsverzeichnis

7.1 Ablauf der Reduktion eines Graphen 62
7.2 Vollständige Reduktion 63
7.3 Reduktion mit zyklenfreien Bereichen 63
7.4 Aufrufgraph des Beispiels 77

Tabellenverzeichnis

4.1 Übersicht über Kategorie 1: Internetprogramme 20
4.2 Übersicht über Kategorie 2: Kommandozeilenprogramme 21
4.3 Übersicht über Kategorie 3: Spiele 21
4.4 Übersicht über Kategorie 4: Compiler 22
4.5 Übersicht über Kategorie 5: Lernprogramme 22
4.6 Übersicht über Kategorie 6: Grafikprogramme 23
4.7 Übersicht über Kategorie 7: Multimediaprogramme 23
4.8 Übersicht über Kategorie 8: Netzwerkprogramme 24
4.9 Übersicht über Kategorie 9: Programme zur Software Entwicklung 24
4.10 Übersicht über Kategorie 10: OS-Programme 24
4.11 Übersicht über die Anpassungen 26
4.12 Größe der Programme 29

5.1 Ergebnisse von function_pointer_check – Teil 1 37
5.2 Ergebnisse von function_pointer_check – Teil 2 40
5.3 Vergleich DAS mit Ergebnissen aus dieser Arbeit 43
5.4 Betrachtung der Ziele von Funktionszeiger 46

6.1 Ergebnisse von routine_check 56

7.1 Ergebnisse von callgraph_check – Teil 1 70
7.2 Ergebnisse von callgraph_check – Teil 2 75
7.3 Analyse der irreduziblen Rubriken 77

8.1 Ergebnisse von cond_check 84
8.2 Ergebnisse von pointer_check 90
8.3 Analyse der Zeiger auf parallele Konstrukte – Teil 1 91
8.4 Analyse der Zeiger auf parallele Konstrukte – Teil 2 91

1 Einleitung

Da Fehler in Programmen nicht nur ärgerlich, sondern mitunter auch sehr gefährlich sein können, muss versucht werden, diese frühzeitig zu finden und zu eliminieren. Mit Hilfe einer Programmanalyse wird nicht nur versucht, diese Fehler zu erkennen, sondern auch Aussagen über das Verhalten bei der Ausführung von Programmen hinsichtlich unterschiedlicher Faktoren, wie die möglichen Ziele von Zeigern oder die Menge der aufgerufenen Funktionen, zu treffen. Neben der dynamischen Analyse, die auf dem compilierten Code zur Ausführungszeit abläuft, existiert ebenfalls eine statische Programmanalyse, die ausschließlich auf dem Quelltext arbeitet. Bei dieser kann das tatsächliche, dynamische Verhalten des Programms lediglich approximativ berechnet werden. Je genauer die Ergebnisse der statischen Analyse aber sind, desto besser kann der vorliegende Code optimiert werden. Als Beispiel sei hier die Eliminierung von totem Code genannt. Wird eine Funktion niemals aufgerufen, so könnte sie aus dem Quelltext entfernt werden. Kommen aber Funktionszeiger im Programm vor und die statische Analyse kann die möglichen Ziele nicht bestimmen, so kann auch nicht ausgeschlossen werden, dass diese Funktion nicht doch Ziel eines solchen Aufrufs ist. Eine genauere Bestimmung der Funktionszeiger ist zwar möglich, aber mit erheblichen Kosten bei der Übersetzung verbunden. Daher steht man in der Praxis immer vor der Frage wie genau die Algorithmen sein müssen, um den Code bestmöglich zu optimieren, ohne dabei langwierige und unnötige Analysen durchzuführen. Neben der Optimierung trifft dies aber auch auf das aufspüren von Fehlern zu, da es sich bei der statischen Programmanalyse um ein falsifizierendes Verfahren, das je nach der Genauigkeit mit der die Analyse durchgeführt wird einen Teil der Fehler finden, aber nicht ihre Abwesenheit garantieren kann.[1]
Einen solchen schnellen und genauen Compiler für alle existierenden Programme zu entwickeln ist allerdings nicht möglich, da die benötigte Genauigkeit der Analyse immer von den im Code verwendeten Programmiermuster abhängt. Ziel dieser Studienarbeit sollte es nun sein, verschiedene Programme aus der Praxis in unterschiedliche Kategorien einzuteilen und zu versuchen, für diese Kategorien eindeutige Charakteristiken zu entdecken. Wenn solche eindeutigen Muster existieren, dann können zumindest für die betrachteten Rubriken Compiler entwickelt werden, die in der Regel auch für weitere Programme aus dieser Rubrik optimal sein sollten. Um dieser empirischen Untersuchung eine möglichst hohe Plausibilität für die Allgemeingültigkeit zu verleihen, sollten mindestens 5 MLoc untersucht werden. Da die verwendeten Programmiermuster vermutlich auch stark von der Sprache des Quellcodes abhängen und die statische Programmanalyse ein sehr weites Feld ist, beschänkt sich diese Arbeit auf Programme die in C oder C++ geschrieben wurden und es werden lediglich Fragestellungen aus den Bereichen der Funktionszeiger-

[1] vgl. Ronen et al. [2006]

Analysen (Wie viele Funktionszeiger gibt es, mit wie vielen Aufrufzielen? Was erkennt die Zeigeranalyse, wo überschätzt sie? ...), der (Kontroll-) Strukturen und Aufrufbeziehungen (Wie viele Unterprogramme besitzen keine oder nur eine Verzweigung? Wie viele sind reine Delegationen? ...), der interprozeduralen Analysen (Ist der Aufrufgraph reduzibel? Welche Tiefe haben CFGs und Aufrufgraphen? ...) und der Zeiger- und Datenflussanalysen (Wie viele Verzweigungen testen einen Zeiger auf NULL oder ungleich NULL? Welche Programmiermuster treten im Umgang mit Zeigern auf Threads und Semaphore, Locks und Condition Variables auf? ...) bearbeitet.
Da eine Analyse einer solch großen Datenmenge von Hand nahezu unmöglich ist, konnte zur Bestimmung der Kenngrössen auf die Werkzeuge des Bauhaus-Projekts zurückgegriffen werden, wobei im Vorfeld für einige der Fragestellungen erst noch passende Analysetools entwickelt werden mussten.
Diese Studienarbeit ist folgendermaßen gegliedert: In Kapitel 2 soll erst einmal ein Überblick über das Bauhaus Projekt, zu welchem Zweck es erstellt wurde und wie es für diese Arbeit genutzt werden konnte gegeben werden. Welche Fragestellungen hier bearbeitet wurden und zu welchem Ziel dies geschah, soll in Kapitel 3 geklärt werden. Die untersuchten Programme und ihre Einteilung in Rubriken werden in Kapitel 4 genannt. Die Kapitel 5 bis 8 beschäftigen sich mit jeweils einer der vier Analysen (Funktionszeiger-Analysen, (Kontroll-) Strukturen und Aufrufbeziehungen, interprozeduralen Analysen und Zeiger- und Datenflussanalysen). Jedes dieser Kapitel teilt sich in die drei Bereiche „Bearbeitung“ (Vorstellung der grundsätzlichen Idee, die hinter dem angefertigten Analyseprogramm steckt, ohne jedoch näher auf den Quellcode einzugehen), „Analyseergebnis“ (Wiedergabe der durch die Analysetools ermittelten Ergebnisse in einer aggregierten Form) und „Auswertung“ (mögliche Verbesserungen an der statischen Programmanalyse anhand der gewonnen Daten werden erörtert). In Kapitel 9 soll in einer kurzen Abschlussbetrchtung die wichtigsten Erkenntnisse aus den gewonnenen Daten zusammengefasst werden und mögliche Schlüsse für die statische Programmanalyse daraus gezogen werden. Da das Hauptaugenmerk dieser Arbeit aber auf der Gewinnung der Charakteristiken der einzelnen Programmrubriken liegt, fällt der letzte Punkt mit den Verbesserungsvorschläge sehr kurz aus.

2 Das Bauhaus Projekt

2.1 Hintergrund

Da ein Großteil der Kosten für Software nicht in der Entwicklung, sondern durch Wartungsarbeiten und Weiterentwicklungen verursacht werden, wäre es wünschenswert Werkzeuge nutzen zu können, die helfen vorhandenen Quellcode zu analysieren und mögliche Fehlerquellen zu erkennen. Gerade bei kritischen Systemen können unbedachte – scheinbar unbedeutende – Änderungen katastrophale Folgen haben.[1]
Die Universität Stuttgart und die Universität Bremen entwickelten zu diesem Zweck gemeinsam das Bauhaus-Projekt – eine Wekzeugsammlung, die zum einen versucht, den Aufbau der Software möglichst genau offen zu legen und zu erklären und zum anderen Analysen und Techniken bereitstellt, mit denen der Code auf Fehler überprüft werden kann oder die Qualität der Programme verbessert wird. Dabei wird das System als ganzes betrachtet und nicht in einzelne Einheiten unterteilt. Momentan unterstützt das Bauhaus-Projekt Programme, die in Ada, C, C++ oder Java geschrieben wurden. Bauhaus selbst wurde hauptsächlich – genau wie alle für diese Studienarbeit benötigten Tools – in Ada geschrieben.
Um die Analyseprogramme aber überhaupt nutzen zu können, müssen die zu testenden Programme erst einmal in eine sinnvolle Darstellung überführt werden. Von dieser hängt die Effizienz und Effektivität jedes Tools im Wesentlichen ab. Wie die optimale Darstellung aber aussieht, ergibt sich allein aus den benötigten Informationen. So nutzen einige Analysen systemnahe Informationen, während anderen eher grobkörnige Informationen, diese jedoch auf einem deutlich höheren Level, erwarten.
Das Bauhaus-Projekt unterstützt beide Darstellungsarten. Um aber den Speichebedarf der Informationen so gering wie möglich zu halten, existieren zwei von einander unabhängige Dateiformate: Die IML, mit detailgetreuen Angaben auf syntaktischer und semantischer Ebene, und die RFG (**R**esource **F**low **G**raph), die einen groben Überblick über den Aufbau des analysierten Systems vermittelt.[2] Für diese Studienarbeit konnte allerdings auf die RFG gänzlich verzichtet werden.

2.2 Die IML

IML steht für **I**nter**M**ediate **L**anguage. Für alle in einer Programmiersprache enthaltenen Elemente wurden Klassen erzeugt, die Methoden zur Verfügung stellen, um weitere Informationen über das vorliegende Element zu erhalten. So lässt sich z.B. für alle Elemente

[1] vgl. Koschke and Plödereder [1996]
[2] vgl. Raza et al. [2006]

der Klasse *C_For_Loop* – also für alle aus der Sprache C stammenden *for*-Schleifen – problemlos die Abbruchbedingung ermitteln. Diese Klassen sind jedoch zusätzlich hierarchisch angeordnet. Das heißt, dass alle Elemente dieser Klasse *C_For_Loop* zusammen mit allen anderen Schleifentypen auch in der übergeordneten Klasse *Loop_Statement* liegen, die ebenfalls weitere Methoden zur Vefügung stellen kann. Somit wird sichergestellt, dass allgemeingültige Methoden nur einmal implementiert werden, während lediglich wenige, speziell auf die jeweilige Sprache angepasste Funktionen in abgeleiteten Kindklassen neu erstellt werden müssen.
Um aus C oder C++ Quellcode eine IML zu erzeugen, existieren sowohl die reinen IML-Compiler *cafeCC* und *cafe++* als auch die Skripte *imlCC* und *imlCXX*, die gleichzeitig eine normale Übersetzung mittels *gcc* und textitcafeCC (bzw. *cafe++*) ermöglichen.[3]
Den Analyse-Programmen die im Rahmen dieser Arbeit erstellt wurden, lag die IML in der Version 24397 zu Grunde.

2.3 Verwendete Tools aus dem Bauhaus Projekt

Neben den bereits erwähnten *cafeCC* und *cafeCXX* wurden in dieser Studienarbeit folgende weitere Tools verwendet:

cobra & iml2html: Gerade bei der Programmierung zusätzlicher Analysen ist es häufig von Nöten, eine Vorstellung von dem zu analysierenden Programms, bzw. von dessen Repräsentation als IML-Graph zu erlangen. Dafür stellt das Bauhaus Projekt zum einen den Code Browser *cobra* und zum anderen *iml2html*, das die Inhalte der IML in einer HTML Darstellung wiedergibt, zur Verfügung. Beide Tools gestatten es, mehr oder weniger Komfortabel, zwischen dem Quellcode und der IML hin und her zu springen und den Graphen zu durchlaufen. Während *cobra* direkt auf der IML agieren kann und auch erheblich mächtiger ist, ermöglicht *iml2html* durch das Trägermedium HTML eine Plattformunabhängigkeit.

das_callgraph & das_tool: Eine Zeigeranalyse nach dem Verfahren von *DAS*[4] kann mit *das_tool* angestoßen werden. Dabei ist zu beachten, dass dieses Verfahren im worst case quadratisch werden kann, was sich bei einem bereits sehr großen IML-Graphen erheblich auf die Berechnungszeit auswirkt. Hierfür kann mittels comand line argument - *undirected* die eigentlich über eine Stufe gerichtete Analyse auf eine ungerichtete *Steensgaard*-Analyse[5] zurückgestuft werden, um so eine beinahe lineare Komplexität zu erzielen. Auch die Strukturanalyse kann auf Wunsch abgeschaltet werden.
das_callgraph fand bei der Analyse des Aufrufgraphen Verwendung, um im Vorfeld alle virtuellen Aufrufe auflösen zu können.

iml2cfg & iml2cg: Neben der normalen IML wurde in dieser Arbeit auch das CFG-Format benötigt, welches den IML-Graphen um weitere Felder anreichert. So wer-

[3] vgl. Raza et al. [2006]
[4] vgl. Das [2000]
[5] vgl. Steensgaard [1996]

den erst nach der Transformation der IML mittels *iml2cfg* Informationen wie die Menge aller aus einer Routine heraus aufgerufenen Routinen, die Beschaffenheit der Grundblöcke und weitere Kontrollflussinformationen ermittelt. Dies ist wichtig, um einen Aufrufgraph aufbauen und Analysen darauf durchführen zu können. Das Tool *iml2cg* kann die Anzahl der möglichen Ziele indirekter Aufrufe bereits im Vorfeld erheblich einschränken und so die Transformation erleichtern.

imlmetrics & sloccount: Mit *imlmetrics* kann eine Vielzahl von Metriken eines Programms berechnet und angezeigt werden. Neben der *Halstead*-Metrik (ein von Maurice Howard Halstead vorgestelltes Maß zur Bestimmung der Komplexität von Software anhand des Verhältnisses von z.B. Variablen zu Vergleichsoperatoren)[6] und der *McCabe*-Metrik (eine Komplexitätsmessung für Software von Thomas McCabe, die die Programme über die Anzahl ihrer Verzweigungen vergleicht)[7] können viele weitere Maßzahlen, wie z.B. die Anzahl der Units, die Anzahl der im Code vorkommenden Routinen, die Anzahl der direkten und indirekten Aufrufe oder die Anzahl der Zuweisungen ermittelt werden. Für die in dieser Arbeite durchgeführten Analysen genügte allerdings die Ausgabe der LOC, um die Größe des bereits bearbeiteten Quellcodes bestimmen zu können.
Während *imlmetrics* diese benötigten Werte aus dem IML Graphen berechnet, also eine Maßzahl liefert aus wie vielen Zeilen das compilierte Programm besteht, gibt *sloccount* wieder aus wie vielen Zeilen der zu Grunde liegende Quelltext besteht.

[6] vgl. Bailey and Dingee [1981]
[7] vgl. McCabe [1976]

3 Aufgabenstellung

In dieser Arbeit werden die Programme aus dem Testrepository hinsichtlich diese vier Bereich der statischen Analyse untersucht:

- Funktionszeigeranalyse
- (Kontroll-) Strukturen und Aufrufbeziehungen
- Interprozedurale Analyse
- Zeiger- und Datenflussanalyse

Da dies sehr weit gefasste Bereiche sind, können natürlich nur bestimmte Teilaspekte untersucht werden. Welche Fragestellungen jeweils untersucht wurden und die möglichen Ansatzpunkte zur Verbesserung der statischen Programmanalyse, die sich aus den erzielten Ergebnissen ergeben könnten, sollen im Folgenden kurz dargestellt werden.

3.1 Funktionszeigeranalyse

Im Bereich der Funktionszeigeranalysen stellt sich die Frage, ob sich der Aufwand einer array- oder feldsensitiven Analyse – also einer Analyse, die nicht nur ein Array bzw. eine Struktur als ganzes betrachtet, sondern auf jedes einzelne Element bzw. Feld eingeht – lohnt, um nachfolgende Analysen, die auf diese Ergebnisse aufbauen, zu beschleunigen oder zu verbessern.
Um diese Entscheidung treffen zu können, benötigt man, neben den Informationen wie viele Funktionszeiger in dem zu untersuchenden Programm vorkommen und auf wie viele Funktionen diese verweisen, natürlich auch eine Einteilung dieser Funktionszeiger in „normale“ Zeiger, Zeiger in Arrays und Zeiger in Strukturen. Mit diesen Werten kann ein Abgleich mit den Ergebnissen einer herkömmlichen Zeigeranalyse stattfinden, um so zu ermitteln wie, genau oder ungenau diese Analysen in der Praxis sind.

3.2 (Kontroll-) Strukturen und Aufrufbeziehungen

„Inlining“ und „Ausrollen“ sind zwei Möglichkeiten zur Optimierung von Quellcode, die häufig bei statischen Analysen zum Einsatz kommen. Dabei wird die Struktur des Quellcodes verändert, indem Teile des Codes mehrfach eingefügt werden, um die Anzahl der Verschmelzungspunkte zu verringern und damit die Genauigkeit von Analysen, die auf dem Kontrollflussgraphen arbeiten, zu erhöhen. Gleichzeitig wird der Quellcode allerdings aufgebläht. Daher stellt sich die Frage, ob sich diese Veränderungen auszahlen.

Da dies vom Aufbau der Unterprogramme abhängt, muss für die Entscheidung, ob sich „Ausrollen“ lohnt, zunächst bestimmt werden, wie stark verzweigt die Unterprogramme sind und bei der Entscheidung, ob „Inlining“ sinnvoll ist, helfen Informationen über die Häufigkeit mit der Aufrufe und Anweisungen in Unterprogramme auftreten. Ebenfalls wird in diesem Zusammenhang geprüft, wie viele der Unterprogramme reine Delegationen sind.

3.3 Interprozedurale Analyse

Zyklen stellen häufig ein Problem bei der Traversierung eines Graphen dar. Bei einer interprozeduralen Analyse sollte man daher wissen, ob der Aufrufgraph eine „unschöne“ Struktur besitzt, also ob in diesem viele solcher Problemstellen vorkommen, oder ob die enthaltenen Zyklen so klein und selten sind, dass Strategien von lokaler Analysen auf Kontrollflussgraphen angewandt werden können.
Zu diesem Zweck wird in Rahmen dieser Studie analysiert, ob die Aufrufgraphen der Testprogramme reduzibel sind, welche Tiefe sie haben und wie groß und häufig eventuelle Zyklen im Graphen sind.

3.4 Zeiger- und Datenflussanalyse

Die bearbeiteten Bereiche der Zeiger- und Datenflussanalyse lassen sich in zwei Teilgebiete einteilen. Zum einen wurden verschiedene Fragestellungen zu Zeiger in Conditions bearbeitet, zum anderen beschäftigt sich diese Arbeit mit Zeigern auf parallele Konstrukte wie Threads, Semaphore, Mutex Locks und Condition Variables.

3.4.1 Zeiger in Conditionals

Eine häufige Fehlerquelle in Programmen stellen Zeigerverwendungen von NULL-Zeiger dar. Diese könnten sehr einfach vom Programmierer abgefangen werden, indem vor dem Zugriff auf einen Zeiger abgeprüft wird, ob das Ziel des Zeigers NULL ist. Wird diese Überprüfung nicht vom Programmierer übernommen, so wäre es hinsichtlich der Sicherheit von Software ratsam, neben einer billigen Zeigeranalyse eine präzise NULL-Zeiger-Analyse bei der statischen Programmanalyse zu verwenden.
Selbstverständlich stellt dies ein gewaltiger Mehraufwand dar, der sich nur bezahlt machen wird, wenn die sicherheitsrelevanten Überprüfungen nicht bereits im Quellcode implementiert wurden. Daher wird in dieser Studie berechnet wie viele Verzweigungen einen Zeiger auf NULL oder ungleich NULL testen und diese Anzahl wird mit der Gesamtzahl der Verzweigungen und Zeigerverwendungen in Relation gesetzt.

3.4.2 Zeiger auf parallelen Strukturen

Da im Zusammenhang mit parallele Konstrukte wie Threads, Semaphore, Mutex Locks und Conditional Variables unter C und C++ in der Regel Zeiger auftreten, stellt sich die Frage, ob eine teure Zeigeranalyse, die speziell auf die Untersuchung von Parallelität in

Programmen angepasst wurde, Sinn macht.
Daher prüft diese Studie wie häufig Zeigern auf Threads, Semaphore, Mutex Locks und Conditional Variables in den Programmen des Testrepositorys auftreten und wie diese verwendet werden, also ob diese Verwendung dem Standard entspricht (geschützter Bereich wird betreten – Aktion wird ausgeführt – Bereich wird verlassen) oder ob ein komplexes Programmiermuster vorliegt.

4 Testprogramme

Im Folgenden sollen alle bei dieser Arbeit verwendeten Testprogramme kurz vorgestellt werden und – soweit verfügbar – die Herkunft des Quellcodes genannt werden. In der Zusammenfassung werden auch die Kategorien, in die diese Programme eingeteilt wurden, genannt. Dabei kann es vorkommen, dass ein Programm in mehrere Gruppen passen könnte. Als Beispiel sei ein Programm genannt, dass sowohl Bild anzeigen, als auch Filme wiedergeben kann. Diese Programme wurden der Kategorie zugeteilt in die sie am besten passen und nicht allen Kategorien, die in Frage kommen. Bei der Auswahl der passenden Kategorie orientierte sich diese Arbeit an den Angaben die der Autor des Programms auf Sourceforge, bzw. auf der Produktseite getätigt hat. Die aus diesen Informationen resultierenden zehn Übergruppierungen sind:

1. Internetprogramme
2. Kommandozeilenprogramme
3. Spiele
4. Compiler
5. Lernprogramme
6. Grafikprogramme
7. Multimediaprogramme
8. Netzwerkprogramme
9. Programme zur Software Entwicklung
10. OS-Programme

Die 101 Testprogramme setzen sich aus 58 reinen C-Programmen, 22 reinen C++-Programmen und 21 C++-Programmen, die zum Teil auch aus C Code bestehen, zusammen. Zum Großteil stammen sie von `http://www.sourceforge.net` oder wurden dem GNU-Projet entnommen. Daher gelten alle gewonnenen Erkenntnisse zunächst nur für quelloffene Software. Bei kommerziellen Produkten könnte es zu einer starken Abweichung von den gesammelten Kenngrößen kommen. Allerdings ist es sehr schwer, an den Quellcode großer kommerzieller Projekte zu gelangen und diesen zur Analyse offen zu legen.
Bei der Auswahl der Programme, die sich noch nicht im Bauhaus Testrepository befanden, wurde darauf geachtet, dass sie sich durch eine relativ gute Bewertung seitens der

Nutzer und durch eine hohe Aktivitätsrate seitens der Entwickler auszeichnen. So sollte sichergestellt werden, dass nur Programme, die in der Praxis auch wirklich eingesetzt werden, in den Analysen auftauchen. Außerdem spricht eine rege Beteiligung der Programmierer an dem Projekt dafür, dass der Code sehr reif ist, damit die Ergebnisse nicht durch stümperhafte Programmierweise verfälscht werden.
In der ersten Rubrik wurden folgende Programme zusammengefasst:

Name	Version	Sprache	Beschreibung	Quelle
apache HTTP Server	1.3.0	C	HTTP (Web-) Server	S.97 Q1
aria2	0.11.1+1	C++	Web und BitTorrent Downloadtool	S.97 Q2
Cntlm	0.31	C	HTTP Proxy	S.97 Q3
CTorrent	1.3.4	C/C++	BitTorrent Client	S.97 Q4
DarkHTTPD	1.2	C	HTTP Server	S.97 Q5
define	0.1p1	C	Client für DICT Anfragen	S.97 Q6
easyhttpd	1.0.1a	C++	C++ Klasse eines HTTP Servers	S.97 Q7
Expat XML Parser	2.0.1	C	XML Parser	S.97 Q8
gSOAP Toolkit	2.7.9k	C/C++	Programme für SOAP Web Services	S.97 Q9
HTML Tidy	4aug00	C	Korrekturhilfe für Webseiten	S.97 Q10
LamaHub	0.0.5.5	C++	NMDC Protokoll Server	S.97 Q11
Licq	1.3.4	C/C++	Instant Messenger	S.97 Q12
mpop	1.0.10	C	POP3 Client	S.97 Q13
msmtp	1.4.12	C	SMTP Client	S.97 Q14
MyServer	0.8.9 rc3	C++	Web Server	S.97 Q15
udt	3.3	C++	Datentransfer Protokoll	S.97 Q16
Urlgfe	1.0.2	C	Download Manager	S.97 Q17
w3m	0.5.2	C	Browser	S.97 Q18
wget	1.10.1	C	Download Manager	S.98 Q19

Tabelle 4.1: Übersicht über Kategorie 1: Internetprogramme

Die zweite Rubrik umfasst diese Programme:

Name	Version	Sprache	Beschreibung	Quelle
BASH	3.1	C	Kommandozeilen Interpreter	S.98 Q20
bc	1.06	C	Kommandozeilenrechner	S.98 Q21
DAR – Disk ARchive	2.3.4	C++	Backup Tool	S.98 Q22

Fortsetzung auf der nächsten Seite...

Fortsetzung Kategorie 2

Name	Version	Sprache	Beschreibung	Quelle
encrypt	0.8	C	Kryptographie Tool	S.98 Q23
grep	2.5.1a	C	Dateisuche auf der Kommandozeile	S.98 Q24
Radmind	1.9.0	C	Kommandozeilen Tools Sammlung	S.98 Q25
random Filesplitter	0.0.2	C++	Programm zum Aufteilen von Dateien	S.98 Q26
screen	4.0.2	C	Terminal Multiplexer	S.98 Q27
tar	1.16	C	Archivierungsprogramm	S.98 Q28
TCSH	6.15.00	C	UNIX C Shell	S.98 Q29
UnZip	5.52	C	Archivierungsprogramm	S.98 Q30

Tabelle 4.2: Übersicht über Kategorie 2: Kommandozeilenprogramme

Folgende Spiele wurden untersucht:

Name	Version	Sprache	Beschreibung	Quelle
Alchera – Game Editor	0.3.1	C/C++	Editor für „Mokoi Gaming“	S.98 Q31
Codebreaker	1.2.1	C	Logik Spiel	S.98 Q32
Corewars	0.9.13	C	Simulation	S.98 Q33
dopewars	1.5.12	C	Simulation	S.98 Q34
Freeciv	2.1.0-b4	C	Rundenbasiertes Strategiespiel	S.98 Q35
glBSP	2.24	C/C++	Editor für die „DOOM“ Engine	S.98 Q36
GNU Go	3.6	C	Umsetzung von „Go“	S.98 Q37
jathena	2060	C	Server für „Ragnarok Online“	S.98 Q38
Pioneers	0.11.1	C	Umsetzung von „Siedler von Catan“	S.98 Q39
Qspacehulk	1.5 b1	C/C++	Umsetzung von „Space Hulk“	S.98 Q40
Qtads	1.6c	C/C++	Interpreter für „Tads“ Spiele	S.98 Q41
SLASH'EM	0.0.8E0F1	C/C++	Abwandlung von „Nethack“	S.98 Q42

Tabelle 4.3: Übersicht über Kategorie 3: Spiele

In der Kategorie Compiler befinden sich diese Programme:

Name	Version	Sprache	Beschreibung	Quelle
Bison	2.3	C/C++	Parser Generator	S.98 Q43
CompBenchmarks	0.5.0	C++	Benchmark für Compiler	S.98 Q44
Concepts	0.3f	C	Sprachwerkzeug	S.98 Q45
Jikes	1.22	C++	Java Compiler	S.99 Q46
Make	3.75	C	Hilfsprogramm für Compiler	S.99 Q47
SWIG	1.3.31	C/C++	Erzeugt Wrapper für C/C++ Code	S.99 Q48
tcc	0.9.23	C	Compiler	S.99 Q49

Tabelle 4.4: Übersicht über Kategorie 4: Compiler

Dies sind die Programme der fünften Rubrik:

Name	Version	Sprache	Beschreibung	Quelle
AstroLib	1.0	C++	Astro Tool	S.99 Q50
Hunspell	1.1.12	C++	Sprachwerkzeug	S.99 Q51
MUMmer	3.20	C/C++	Abbildung von genetischer Information	S.99 Q52
muparser	128	C++	Mathematik Parser	S.99 Q53
Open Dynamics Engine	0.8	C/C++	Simulation von dynamischen Körpern	S.99 Q54
QUCS	0.0.12	C++	Stromkreislauf Simulation	S.99 Q55
stardict	2.4.8	C++	Wörterbuch	S.99 Q56
TextMatrix	1.0	C++	Text Mining	S.99 Q57
units	1.86	C	Mathematischer Konverter	S.99 Q58

Tabelle 4.5: Übersicht über Kategorie 5: Lernprogramme

Als Vertreter von Grafikprogrammen wurden folgende Tools gewählt:

Name	Version	Sprache	Beschreibung	Quelle
AFPL Ghostscript	8.54	C	PS und PDF Interpreter	S.99 Q59
Dia	0.95.0	C++	Visualisierungstool	S.99 Q60
Gimp	2.2.13	C	Vielseitiges Grafiktool	S.99 Q61
GLE	4.0.12	C/C++	Visualisierungstool	S.99 Q62
GNU Ghostscript	8.01	C	PS und PDF Interpreter	S.99 Q63
GNU Plot	4.0.0	C	Plotting Tool	S.99 Q64
GQview	2.0.1	C	Bildbetrachter	S.99 Q65

Fortsetzung auf der nächsten Seite...

Fortsetzung Kategorie 6

Name	Version	Sprache	Beschreibung	Quelle
JLRFractal	1.0	C++	Mandelbrot Generator	S.99 Q66
Pixie	2.2.2	C/C++	3D Rendersoftware	S.99 Q67
POV-Ray	3.6.1	C++	3D Rendersoftware	S.99 Q68
Roadmap	1.0.12	C	Navigationssystem	S.99 Q69
RRDtool	1.2.23	C/C++	Erstellung von Schaubildern	S.99 Q70
Sodipodi	0.34	C	Vektorbasiertes Grafikprogramm	S.99 Q71
trueprint	5.3	C	PS Drucker	S.99 Q72
xpdf	3.01	C++	PDF Betrachter	S.100 Q73

Tabelle 4.6: Übersicht über Kategorie 6: Grafikprogramme

Testprogramme der Rubrik Multimediaprogramme:

Name	Version	Sprache	Beschreibung	Quelle
AlsaPlayer	0.99.80-rc2	C/C++	CD Player	S.100 Q74
Aqualung	0.9 b8	C/C++	Musikspieler	S.100 Q75
SMPlayer	0.5.0	C++	Front-End für den MPlayer	S.100 Q76
SoundTracker	0.6.7	C	Musikeditor	S.100 Q77
Streamripper	1.62.1	C	Aufnahmeprogramm	S.100 Q78
transfig	3.2.5	C	Converter	S.100 Q79

Tabelle 4.7: Übersicht über Kategorie 7: Multimediaprogramme

Diese Testprogramme zählen zur Kategorie 8:

Name	Version	Sprache	Beschreibung	Quelle
Clusterit	2.4	C	Sammlung von Clustering Tools	S.100 Q80
FireStarter	1.0.3	C	Firewall	S.100 Q81
GGMud	0.7	C	mud Client	S.100 Q82
glBox	0.1	C	Logging Tool	S.100 Q83
iometer	2006-07-27-RC3	C/C++	Benchmark	S.100 Q84
Nagios	3.0a4	C	Monitoring Tool	S.100 Q85
SNNS	4.2	C	Simulation neuronaler Netze	S.100 Q86

Fortsetzung auf der nächsten Seite...

Fortsetzung Kategorie 8

Name	Version	Sprache	Beschreibung	Quelle
time	1.7	C	Überblick über Resourcenverbrauch	S.100 Q87
uucp	1.07	C	Freie UUCP Software	S.100 Q88
zabbix	1.3.8	C	Monitoring Tool	S.100 Q89

Tabelle 4.8: Übersicht über Kategorie 8: Netzwerkprogramme

Als Programme zur Software Entwicklung wurden diese Tools ausgewählt:

Name	Version	Sprache	Beschreibung	Quelle
Bluefish	1.0.7	C	Webprogrammierung	S.100 Q90
cscope	15.6	C	Sourcecode Browser	S.100 Q91
Gwenhywfar	1.19.2	C	Portierungstool	S.100 Q92
JOE	3.5	C	Texteditor	S.100 Q93
nano	1.2.3	C	Texteditor	S.100 Q94
sed	4.1	C	Streameditor	S.100 Q95

Tabelle 4.9: Übersicht über Kategorie 9: Programme zur Software Entwicklung

Diese OS-Programme komplettieren das Testrepository:

Name	Version	Sprache	Beschreibung	Quelle
HT editor	2.0.10	C/C++	Betrachter für ausführbare Dateien	S.100 Q96
keyTouch	2.3.1	C	Belegung zusätzlicher Tasten	S.100 Q97
OProfile	0.9.3	C/C++	Profilingsystem	S.100 Q98
ScramDisk 4 Linux	1.1.0	C/C++	Zugriff auf SD und TC Container	S.100 Q99
synergy	1.3.1	C++	Nutzt eine Maus für mehrere PCs	S.100 Q100
Xplorer filemanager	0.7.0	C++	Dateimanager	S.100 Q101

Tabelle 4.10: Übersicht über Kategorie 10: OS-Programme

Die meisten Programme lassen sich einfach in die IML bringen, indem man die Variablen *CC*, *CXX*, *AR*, *LD* und *LN* auf *imlCC*, *imlCXX*, *imlAR*, *imlCC* bzw. *imlCXX* und *imlLN* umbiegt. Dies kann entweder direkt durch die Übergabe an *configure*, also z.B. „*./configure CC=imlCC CXX=imlCXX AR=imlAR LD=imlCC LN=imlLN*“, geschehen oder bei bereits erzeugten Makefiles über „*make 'CC = imlCC' 'CXX = imlCXX' 'AR = imlAR' 'LD = imlCC' 'LN = imlLN'*“. In seltenen Fällen führt dies aber nicht zu korrekt erzeugten IML Dateien, da entweder die gesetzten Variablen nicht verwendet werden, also z.B. anstelle „*CC foo.c*“ steht im Makefile „*gcc foo.c*“ und es findet nur eine

normale Übersetzung statt, oder die im Makefile angegebene Ausführreihenfolge nicht stimmt.

4.1 Anpassungen der Makefiles

Damit diese Programme dennoch analysiert werden können, müssen die Makefiles von Hand angepasst und die Vorkommnisse von *gcc, ar, ln, ...* müssen durch *imlCC, imlAR, imlLN, ...* ersetzt werden oder die Makefiles müssen in einer anderen Reihenfolge ausgeführt werden, als darin angegeben wird. Damit die Ergebnisse nachvollzogen werden können, werden im Folgenden die Anpassungen an den Programmen genannt.

Name	Änderung
Alchera – Game Editor	*CPP = imlCXX* in *alchera-0_3_1/Alchera/linux.make* und *alchera-0_3_1/Alchera/makefile-base* setzen.
apache HTTP Server	In allen Makefiles *ar* von Hand ersetzen (z.B. *apache_1.3.0/src/ap/Makefile* Zeile 59).
define	*GCC = imlCC* im Makefile setzen.
easyhttpd	In *build.sh* die Übersetzung mit *imlCXX* durchführen.
GGMud	In allen Makefiles *ar* von Hand ersetzen (z.B. *ggmud/Makefile* Zeile 84). Bevor *make* ausgeführt werden kann zunächst *ggmud/zlib/make linz.a* ausführen.
jathena	In Makefile *CC = imlCC -pipe* setzen.
ScramDisk 4 Linux	In allen Makefiles alle Vorkommnisse von *ar, gcc, ...* von Hand mit *imlAR, imlCC, ...* ersetzen, dabei aber alle Parameter beibehalten.
SMPlayer	In *smplayer-0.5.0/src/Makefile LINK = imlCXX* und *AR = imlAR cqs* setzen und diese ausführen.
TextMatrix	In allen Makefiles *ar* von Hand ersetzen (z.B. *TextMatrix.v1.0/src/Makefiles/Makefile.LibTextMatrix* Zeile 29).
transfig	In *transfig.3.2.5/Makefile, transfig.3.2.5/fig2dev/Makefile, transfig.3.2.5/fig2dev/dev/Makefile* und *transfig.3.2.5/transfig/Makefile* alle Vorkommnisse von *ar, gcc, ...* von Hand mit *imlAR, imlCC, ...* ersetzen, dabei aber alle Parameter beibehalten.
udt	In allen Makefiles *ar* von Hand ersetzen (z.B. *udt3/src/Makefile* Zeile 53) und *C++* auf *imlCXX* umsetzen.

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Änderung
xpdf	In Makefiles *AR = imlAR rc* setzen.
zabbix	*zabbix-1.3.8/src/zabbix_agent/make*, *zabbix-1.3.8/src/zabbix_get/make* und *zabbix-1.3.8/src/zabbix_sender/make* zur Übersetzung ausführen.

Tabelle 4.11: Übersicht über die Anpassungen

Eine weiter Studienarbeit, die alle Vorkommnisse *ar, gcc, ...* automatisch auf *imlAR, imlCC, ...* umbiegt, könnt diese Anpassungen obsolet machen, aber das Tool war zur Zeit der Übersetzung der oben genannten Programme noch nicht verfügbar.[1]

Eine weiter Fehlerquelle, die zu Übersetzungsfehlern führen kann, stellt die zum Teil gegebene Inkompatibilität von Bauhaus mit der *gcc* Version 4.2 dar. Daher sollte bei der Compilierung nach Möglichkeit auf eine ältere Version von *gcc* zurückgegriffen werden.

4.2 Größe der Programme

Neben der Kategorisierung der Testprogramme ist auch ihre Größe für die folgenden Analysen entscheidend. Dafür werden einerseits die lines of code (TLOC) betrachtet, die *imlmetrics* anhand des IML Graphen berechnen konnte (inklusive aller verwendeter Bibliotheken) und andererseits die von *sloccount* ermittelten Zeilen Quellcode (SLOC), die zur Compilierung nötig waren (inklusive aller Kommentare).

Name	TLOC	SLOC
apache HTTP Server	22290(*)	51242
aria2	32946	41420
Cntlm	2507	2914
CTorrent	3859	4883
DarkHTTPD	1714	1616
define	416	348
easyhttpd	19053(*)	769
Expat XML Parser	9813	24610
gSOAP Toolkit	58544(*)	79798
HTML Tidy	9698	11740
LamaHub	17520	6626
Licq	47740	98225
mpop	7574	20191
msmtp	6566	18102
MyServer	54110(*)	42382

Fortsetzung auf der nächsten Seite...

[1] vgl. Berger [2007]

Fortsetzung

Name	TLOC	SLOC
udt	81252(*)	9236
Urlgfe	16207(*)	25313
w3m	255535(*)	67662
wget	17556	29721
Kategorie 1	**664900**	**536798**
BASH	51833(*)	108941
bc	7444(*)	10155
DAR – Disk ARchive	215586(*)	41524
encrypt	425	2748
grep	5489	24818
Radmind	31573(*)	19086
random Filesplitter	3309	113
screen	24845	39310
tar	20512(*)	80438
TCSH	27757(*)	54987
UnZip	12202(*)	54779
Kategorie 2	**400975**	**436899**
Alchera – Game Editor	26586(*)	20062
Codebreaker	2314	1555
Corewars	12659(*)	12397
dopewars	17053	40559
Freeciv	247311(*)	256013
glBSP	4653	17715
GNU Go	98165(*)	158443
jathena	85207(*)	98533
Pioneers	63663(*)	42208
Qspacehulk	43214	31460
Qtads	85568	82016
SLASH'EM	116063(*)	257521
Kategorie 3	**802456**	**1018482**
Bison	14073	80399
CompBenchmarks	41969(*)	36615
Concepts	3093	4100
Jikes	61876	74224
Make	8449	17668
SWIG	53162	111408
tcc	12754(*)	43496
Kategorie 4	**195376**	**367910**

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	TLOC	SLOC
AstroLib	2004	5393
Hunspell	55558	39763
MUMmer	142811(*)	27985
muparser	28568(*)	7921
Open Dynamics Engine	923802(*)	75709
QUCS	193671(*)	132046
stardict	101864(*)	40727
TextMatrix	216817(*)	46912
units	2488	3442
Kategorie 5	**1667583**	**379871**
AFPL Ghostscript	241285(*)	467161
Dia	48988	138110
Gimp	3298372(*)	596836
GLE	115883(*)	57201
GNU Ghostscript	167570(*)	279547
GNU Plot	57210(*)	76458
GQview	49532	53867
JLRFractal	26562	3060
Pixie	68879(*)	467926
POV-Ray	42242	220783
Roadmap	97192(*)	32642
RRDtool	13130(*)	32240
Sodipodi	85076(*)	121825
trueprint	5858	9402
xpdf	190029(*)	74831
Kategorie 6	**4507808**	**2631889**
AlsaPlayer	21833	49037
Aqualung	43029	40675
SMPlayer	40946	16000
SoundTracker	22362	37363
Streamripper	15555(*)	54352
transfig	21162(*)	31650
Kategorie 7	**164887**	**229077**
Clusterit	13110(*)	9824
FireStarter	10434	24700
GGMud	23363	33291
glBox	973	770
iometer	19137	28219

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	TLOC	SLOC
Nagios	163979(*)	74209
SNNS	168228(*)	94434
time	604	1627
uucp	75014(*)	58123
zabbix	17953(*)	86484
Kategorie 8	**492795**	**411681**
Bluefish	27400	43602
cscope	17651	17555
Gwenhywfar	108332(*)	84022
JOE	23712(*)	36643
nano	6608	13364
sed	8987	28343
Kategorie 9	**192690**	**223529**
HT editor	90066(*)	116328
keyTouch	10363(*)	13181
OProfile	169916(*)	28044
ScramDisk 4 Linux	74134(*)	20350
synergy	69537(*)	49384
Xplorer filemanager	53245(*)	31392
Kategorie 10	**467261**	**258679**

Werte mit (*) setzen sich aus mehreren Programmen zusammen

Tabelle 4.12: Größe der Programme

Insgesamt wurden also 9556731 LOC aus IML Dateien (bzw. 6494815 LOC Quellcode) untersucht, was zwar deutlich über dem geforderten Wert von 5MLoc liegt, aber bewusst so hoch gewählt wurde, da in diesen rund 10MLoc auch mehrfach benutzte Bibliotheken enthalten sind und so versucht wurde, den Anteil an neuem, noch nicht untersuchtem Code möglichst hoch zu halten.

5 Funktionszeiger-Analysen

5.1 Bearbeitung

Für die Bearbeitung der Fragestellungen im Bereich der Funktionszeiger-Analysen wurde im Rahmen diese Arbeit das Tool *function_pointer_check* erstellt. In diesem werden in einem ersten Schritt alle nicht künstlichen Variablen, die im Quellcode bzw. in der IML vorkommen, gesucht. Wenn sich diese Variable in der Klasse *T_Pointer* befindet, so wird sie in zwei Listen aufgenommen. In der einen Liste wird lediglich der Knoten der Variable selbst gespeichert. Dies dient nur dazu schnell überprüfen zu können, ob diese Variable bereits in der Liste ist, also schon abgearbeitet ist und lediglich ein neues Auftreten des Zeigers festgestellt wurde oder ob sie neu aufgenommen werden muss. Wenn die Variable in der Liste bereits existiert, hilft die Funktion *Add_Location* dabei diese an der richtigen Stelle einzufügen. Für die zweite, eigentliche Ergebnisliste wurde ein neuer Recordtype *Pointer_Record* eingeführt. In diesem wird neben dem Knoten des Zeigers auch eine Liste mit seinen Zielen, die Anzahl der unterschiedlichen Ziele, seinen Vorkommnissen im Programm, einer *Kill*-Information, auf die an späterer Stelle noch näher eingegangen wird und seine Art gespeichert. Mit der Art ist gemeint, ob es sich bei dem Zeiger um einen „normalen“ Zeiger handelt oder ob sich dieser in einem Array oder einer Struktur befindet.

Diese Unterscheidung des Zeigertyps geschieht in drei Schritten. Im ersten Schritt wird geprüft, ob sich die Variable in der Klasse *T_Pointer* befindet. Dies sind alle „normalen“ Zeiger. Ist dies nicht der Fall, muss in einem zweiten Schritt geprüft werden, ob es sich bei der Variable um eine Instanz der Klasse *T_Array* handelt. Bei Arrays muss natürlich zusätzlich geprüft werden, ob die beinhalteten Elemente Zeiger sind.

Strukturen liegen nicht in der Klasse *O_Variable*. Daher müssen im dritten Schritt alle nicht künstlichen Felder durchlaufen werden und die Klasse der Felder betrachtet werden. Bei Feldern, die Zeiger beinhalten, wird wie oben bereits beschrieben ein neues Element der Ergebnisliste hinzugefügt, bzw. ein bestehendes um Ortsangaben erweitert.

In all diesen Schritten wird zwischen Zeigern im Allgemeinen und Funktionszeigern im Speziellen noch nicht unterschieden. Daher ist bei allen Zeigern *Kill* gesetzt, da diese zu diesem Zeitpunkt noch auf keine Funktion zeigen und daher für das Analyseergebnis keine Rolle spielen. Für diese Unterscheidung werden die Ziele der Zeiger benötigt. Um an diese zu gelangen, werden alle Zuweisungen überprüft, da nur an diesen Stellen das Ziele wechseln kann. Somit werden nur die Funktionszeiger betrachtet, denen wenigstens einmal ein Ziel zugewiesen wurde.

Eine Instanz der Klasse *Assignment* hat ebenfalls das Attribut *Its_Type*, mit dem man durch eine einzige Überprüfung feststellen kann, ob die Zuweisung weiter untersucht werden muss, also ob sie vom Typ *T_Pointer* ist. Ist dies der Fall, muss die linke Seite der

Zuweisung so lange aufgelöst werden, bis man bei der Klasse *O_Node* und somit bei dem Zeiger selbst ist. Dies muss ebenfalls in drei Schritten, analog zur Feststellung des Zeigertyps, geschehen. Bei Arrays wird diese Konvertierung durch die Funktion *Small_Convert* unterstützt.

Um die rechte Seite, also das Ziel des Zeigers, auflösen zu können wurde die Funktion *Convert* erstellt. In dieser werden viele Konvertierungen durchgeführt und geprüft, ob das Ziel in der Klasse *O_Routine* liegt. Nur dann handelt es sich bei dem Zeiger auch um einen Funktionszeiger. Wenn *Convert* festgestellt hat, dass ein Funktionszeiger vorliegt, so kann mittels *Add_Aim* die Routine, die ein Ziel des Zeigers ist, an der richtigen Stelle in der Ergebnisliste eingefügt werden. Wenn der Zeiger auf ein für ihn neues Ziel zeigt, so wird dieses in *R_Count* aufaddiert. Jeder Zeiger, dem mindestens einmal ein Ziel zugeordnet wurde verliert seine *Kill* Information.

Nun befinden sich aber immer noch normale Zeiger in der Hilfsliste. Diese werden mit dem Aufruf *Clear_List* aufgespürt und gelöscht, sollte bei ihrem Gegenstück in der richtigen Ergebnisliste die *Kill*-Flag gesetzt sein. Kommt ein Zeiger an keiner Positionen im Quellcode vor oder besitzt er kein Ziel, so darf er nicht zur Ergebnismenge hinzugerechnet werden. Daher werden diese eventuell „falschen“ Zeiger aus der Hilfsliste gestrichen.

```
int M(void)
{
    return 0
}

int main(void)
{
    int (*fp) (int);
    int (*fp2) (int);

    fp = &M;
    fp2 = fp;

    return 1;
}
```

Bislang hat *function_pointer_check* nur fp entdeckt, aber fp2 nicht, da dieser Zeiger nicht direkt auf eine Funktion zeigt. Um diesen Zeiger auflösen zu können, muss der Graph immer wieder durchlaufen werden, bis keine neue Funktionszeiger gefunden werden. Da dies bei großen Graphen sehr lange dauern kann, wurde die Konstante *Stufe* definiert, die angibt bis zu welcher Tiefe – also wie oft – die Zeiger des Graphen durchsucht werden sollen.

Dafür werden erneut die Zuweisungen im Programm, die auf der linken Seite einen Zeiger haben, durchlaufen. Die rechte Seite wird aber nicht erneut mit *Convert*, sondern mit *Convert2* aufgelöst. Diese Funktion versucht aber nicht bis zu einer Instanz der Klasse *O_Routine* zu konvertieren, da diese ja bereits alle gefunden worden sein sollten, sondern nur so lange bis ein Zeiger gefunden wird. Befindet sich links und rechts der Zuweisung

ein Zeiger (z.B. bei fp2 = fp;), so handelt es sich genau dann um eine Zuweisung an einen Funktionszeiger, wenn der Zeiger der rechten Seite ein Funktionszeiger ist. Diese Überprüfung führt die Funktion *Add_Aim2* aus. Zunächst wird der Zeiger auf der linken Seite in der Liste aller im Programm vorkommender Zeiger gesucht. Wurde er gefunden, so wird gesucht, ob die rechte Seite bereits als Funktionszeiger identifiziert wurde, also ob sich der rechte Zeiger in der Liste der Funktionszeiger befindet und nicht als *Kill* ausgezeichnet ist. Ist dies der Fall, also handelt es sich um einen Funktionszeiger. Dann läuft der Rest von *Add_Aim2* analog zu *Add_Aim* ab. Wieder wird gesucht, ob die Routine ein neues Ziel ist und der Liste aller Ziele hinzugefügt. Zusätzlich wird bei jedem „Treffer“ dem Hauptprogramm mitgeteilt, dass ein neuer Zeiger der Zeigerliste hinzugefügt werden konnte, da dann der gesamte Graph ein weiteres mal durchlaufen werden muss.

Die Ausgabe erfolgt schließlich mittels *Output*. Dabei wird in einer *Comma Separated Values* (CSV)-Datei der Name, der untersuchten IML Datei, die Anzahl aller Funktionszeiger, die Anzahl der „normalen“ Funktionszeiger, die Funktionszeiger in einem Array, die Funktionszeiger in einer Struktur, die minimale und maximale Anzahl der Ziele aller Funktionszeiger im Programm und die Anzahl der Ziele für jeden Zeiger im Programm gespeichert. Hier werden natürlich nur Zeiger betrachtet, bei denen die *Kill*-Flag nicht gesetzt ist.

Sollte die angegebene Ausgabedatei nicht existieren, stellt *Secure_Open* sicher, dass diese zunächst erstellt wird und es zu keiner Ausnahme kommt.

function_pointer_check liegt in $O(m^2 * n)$, wobei m für die Zahl der Zeiger in dem Programm und n für die Anzahl der Zuweisungen steht. Dieser Wert ergibt sich, da bei jeder Zuweisung, die im Programm gefunden wird, zunächst alle Zeiger durchsucht werden, ob diese auf der linken oder rechten Seite der Zuweisung stehen. Die Anweisungen müssen so oft durchlaufen werden, bis keine neuen Funktionszeiger gefunden werden. In einem Programm wie

```
int main (void)
{
...
    fpy = fpx;
...
    fp2 = fp1;
    fp1 = funktion;
...
}
```

tritt der worst case mit $O(m^2 * n)$ wirklich ein, aber in der Praxis liegt das Programm eher in $O(c * m * n)$ mit $c < 10$.

{function_pointer_check: 695 LOC}

5.2 Analyseergebnisse

In den folgenden Tabellen wurden die Ergebnisse der Analyse *function_pointer_check* zusammengefasst. Sowohl bei Programmen, die aus mehreren Unterprogrammen bestehen, als auch bei der Zusammenfassung am Ende jeder Rubrik wurden die Werte der inneliegenden Einheiten aufsummiert, bzw. das Minimum oder Maximum bestimmt.

In jeder Spalte findet man den Namen des Programms, die Anzahl aller Funktionszeiger, die Anzahl der „normalen“ Funktionszeiger, die Anzahl der Funktionszeiger in Arrays und die Anzahl der Funktionszeiger in Strukturen.

Name	FP	NP	IA	IS
apache HTTP Server	846	657	2	187
aria2	88	59	0	26
Cntlm	68	58	0	10
CTorrent	33	30	0	3
DarkHTTPD	52	39	0	13
define	3	3	0	0
easyhttpd	6	4	0	2
Expat XML Parser	123	78	0	45
gSOAP Toolkit	959	791	3	165
HTML Tidy	377	312	4	61
LamaHub	180	155	0	25
Licq	728	550	0	178
mpop	178	147	0	31
msmtp	159	126	0	33
MyServer	185	164	2	19
udt	88	88	0	0
Urlgfe	798	508	1	289
w3m	1168	923	2	243
wget	502	440	0	62
Kategorie 1	**6541**	**5132**	**14**	**1395**
BASH	1737	1569	6	162
bc	152	125	2	25
DAR – Disk ARchive	522	408	0	114
encrypt	16	16	0	0
grep	120	96	0	24
Radmind	534	440	0	94
random Filesplitter	0	0	0	0
screen	412	330	3	79

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	FP	NP	IA	IS
tar	572	529	1	42
TCSH	752	676	8	68
UnZip	102	55	0	47
Kategorie 2	**4919**	**4244**	**23**	**655**
Alchera – Game Editor	738	671	4	63
Codebreaker	57	54	1	2
Corewars	312	231	0	81
dopewars	746	639	14	93
Freeciv	14693	11221	36	3436
glBSP	176	141	1	34
GNU Go	202	170	2	30
jathena	1918	1703	46	169
Pioneers	1866	1416	9	441
Qspacehulk	159	147	0	12
Qtads	954	814	0	140
SLASH'EM	1374	1270	5	99
Kategorie 3	**23195**	**18477**	**118**	**4600**
Bison	392	359	1	32
CompBenchmarks	250	232	0	18
Concepts	96	81	0	15
Jikes	2100	1764	0	336
Make	424	346	3	75
SWIG	6293	6012	3	278
tcc	236	211	0	25
Kategorie 4	**9791**	**9005**	**7**	**779**
AstroLib	12	10	0	2
Hunspell	436	341	6	89
MUMmer	233	200	12	21
muparser	20	16	0	4
Open Dynamics Engine	6205	5757	0	448
QUCS	2429	2118	1	310
stardict	992	847	1	144
TextMatrix	289	213	0	76
units	74	51	4	19
Kategorie 5	**10690**	**9553**	**24**	**1113**
AFPL Ghostscript(*)	1916	970	6	940
Dia(*)	2104	1694	1	409
Gimp(*)	100921	85839	32	15050

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	FP	NP	IA	IS
GLE	534	459	19	56
GNU Ghostscript(*)	1249	611	4	634
GNU Plot	707	619	2	86
GQview	2560	2119	5	436
JLRFractal	18	14	0	4
Pixie	1324	1192	7	125
POV-Ray	398	292	2	104
Roadmap	1592	1271	24	297
RRDtool	177	141	0	36
Sodipodi(*)	3776	2876	1	899
trueprint	50	44	0	6
xpdf	2226	1588	25	613
Kategorie 6	**119552**	**99729**	**128**	**19695**
AlsaPlayer	147	135	0	12
Aqualung	1282	1188	4	90
SMPlayer	103	103	0	0
SoundTracker	637	502	12	123
Streamripper	198	163	0	35
transfig	262	193	1	68
Kategorie 7	**2629**	**2284**	**17**	**328**
Clusterit	341	307	1	33
FireStarter	575	464	0	111
GGMud	944	852	8	84
glBox	30	26	1	3
iometer	33	25	0	8
Nagios	9904	5841	5	4058
SNNS	2049	1779	2	268
time	6	6	0	0
uucp	1496	1283	14	199
zabbix	294	252	0	42
Kategorie 8	**15672**	**10835**	**31**	**4806**
Bluefish	1876	1542	5	329
cscope	141	128	1	12
Gwenhywfar	4811	4307	3	501
JOE	816	617	2	197
nano	245	225	0	20
sed	222	156	0	66
Kategorie 9	**8111**	**6975**	**11**	**1125**

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	FP	NP	IA	IS
HT editor	1374	1129	2	243
keyTouch	359	269	0	90
OProfile	788	708	1	79
ScramDisk 4 Linux	30	22	0	8
synergy	266	247	0	19
Xplorer filemanager	88	70	0	18
Kategorie 10	**2905**	**2445**	**3**	**457**

Werte mit (*) wurden mit einer einstufigen Richtungsanalyse ermittelt

Tabelle 5.1: Ergebnisse von function_pointer_check – Teil 1

In dieser zweiten Tabelle mit den Ergebnissen, die von *function_pointer_check* berechnet wurden, stehen für jedes Programm neben seinem Namen die minimale, die durchschnittliche und die maximale Anzahl der verschiedenen Ziele aller gefundener Funktionszeiger.

Name	Min_Aim	Avg_Aim	Max_Aim
apache HTTP Server	1	1,1	6
aria2	1	1,0	2
Cntlm	1	1,1	3
CTorrent	1	1,0	1
DarkHTTPD	1	1,1	2
define	1	1,0	1
easyhttpd	1	1,0	1
Expat XML Parser	1	1,6	51
gSOAP Toolkit	1	1,1	5
HTML Tidy	1	1,2	10
LamaHub	1	1,0	2
Licq	1	1,1	12
mpop	1	1,1	4
msmtp	1	1,1	3
MyServer	1	1,1	3
udt	1	1,0	1
Urlgfe	1	1,1	3
w3m	1	1,2	7
wget	1	1,1	4
Kategorie 1	**1**	**1,1**	**51**
BASH	1	1,3	16
bc	1	1,1	3

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min_Aim	Avg_Aim	Max_Aim
DAR – Disk ARchive	1	1,0	2
encrypt	1	1,0	1
grep	1	1,3	4
Radmind	1	1,1	3
random Filesplitter	–	–	-
screen	1	1,1	10
tar	1	1,1	3
TCSH	1	1,2	7
UnZip	1	1,0	2
Kategorie 2	**1**	**1,2**	**16**
Alchera – Game Editor	1	1,1	11
Codebreaker	1	1,1	2
Corewars	1	1,0	2
dopewars	1	1,1	3
Freeciv	1	1,2	10
glBSP	1	1,1	2
GNU Go	1	1,1	5
jathena	1	1,1	4
Pioneers	1	1,1	3
Qspacehulk	1	1,1	7
Qtads	1	1,1	9
SLASH'EM	1	1,2	13
Kategorie 3	**1**	**1,2**	**13**
Bison	1	1,1	2
CompBenchmarks	1	1,1	3
Concepts	1	1,1	2
Jikes	1	1,2	8
Make	1	1,3	6
SWIG	1	1,1	6
tcc	1	1,2	6
Kategorie 4	**1**	**1,1**	**8**
AstroLib	1	1,3	4
Hunspell	1	1,2	7
MUMmer	1	1,0	2
muparser	1	1,0	1
Open Dynamics Engine	1	1,1	10
QUCS	1	1,3	14
stardict	1	1,2	5

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min_Aim	Avg_Aim	Max_Aim
TextMatrix	1	1,1	4
units	1	1,1	2
Kategorie 5	**1**	**1,1**	**14**
AFPL Ghostscript(*)	1	1,3	57
Dia(*)	1	1,2	22
Gimp(*)	1	1,1	25
GLE	1	1,3	5
GNU Ghostscript(*)	1	1,3	57
GNU Plot	1	1,9	7
GQview	1	1,3	5
JLRFractal	1	1,0	1
Pixie	1	2,5	6
POV-Ray	1	1,4	4
Roadmap	1	1,4	3
RRDtool	1	1,3	3
Sodipodi(*)	1	1,3	20
trueprint	1	1,3	2
xpdf	1	1,2	8
Kategorie 6	**1**	**1,1**	**57**
AlsaPlayer	1	1,0	2
Aqualung	1	1,0	4
SMPlayer	1	1,0	2
SoundTracker	1	1,1	13
Streamripper	1	1,1	4
transfig	1	1,1	3
Kategorie 7	**1**	**1,1**	**13**
Clusterit	1	1,3	3
FireStarter	1	1,0	3
GGMud	1	1,0	3
glBox	1	1,0	1
iometer	1	1,0	1
Nagios	1	1,0	4
SNNS	1	1,1	3
time	1	1,0	1
uucp	1	1,1	7
zabbix	1	1,1	2
Kategorie 8	**1**	**1,1**	**7**
Bluefish	1	1,1	6

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min_Aim	Avg_Aim	Max_Aim
cscope	1	1,2	3
Gwenhywfar	1	1,2	9
JOE	1	1,1	27
nano	1	1,4	6
sed	1	1,2	6
Kategorie 9	**1**	**1,2**	**27**
HT editor	1	1,1	9
keyTouch	1	1,0	2
OProfile	1	1,1	3
ScramDisk 4 Linux	1	1,2	2
synergy	1	1,0	3
Xplorer filemanager	1	1,0	2
Kategorie 10	**1**	**1,1**	**9**

Werte mit (*) wurden mit einer einstufigen Richtungsanalyse ermittelt

Tabelle 5.2: Ergebnisse von function_pointer_check – Teil 2

Zusätzlich zu den oben genannten Ergebnissen, sollten die Anzahl der berechneten Ziele der Funktionszeiger (RA) mit denen, die von einer *DAS*-Analyse (DA) bestimmt werden, verglichen werden, um festzustellen wie sehr diese Analyse in der Realität überschätzt. Ist die Differenz zwischen den richtigen und den durch die Analyse bestimmten Werten sehr hoch, so wäre eine noch genauere – aber auch teurere – Zeigeranalyse, wie z.B. *Andersen*[1] oder *Wilson*[2], bei der statischen Analyse angebracht.

[1] vgl. Andersen
[2] vgl. Wilson and Lam [1995]

Name	RA	DA
apache HTTP Server	962	4393131
aria2	91	148985
Cntlm	76	6721
CTorrent	33	41982
DarkHTTPD	59	1613
define	3	14
easyhttpd	6	167
Expat XML Parser	192	376419
gSOAP Toolkit	1069	4921487
HTML Tidy	440	129890
LamaHub	185	73775
Licq	769	5493316
mpop	201	33264
msmtp	174	43802
MyServer	196	600362
udt	88	276660
Urlgfe	839	115462
w3m	1363	13235809
wget	562	440989
Kategorie 1	**7306**	**30333848**
BASH	2248	4271217
bc	168	58392
DAR – Disk ARchive	535	126443922
encrypt	16	0
grep	152	50023

Fortsetzung auf der nächsten Seite...

Name	RA	DA
Radmind	576	40490
random Filesplitter	0	19
screen	465	2278657
tar	620	4053934
TCSH	867	2983938
UnZip	105	322908
Kategorie 2	**5752**	**140503500**
Alchera – Game Editor	814	726213
Codebreaker	63	244
Corewars	326	15582
dopewars	805	382872
Freeciv	17396	206973915
glBSP	194	68591
GNU Go	221	57565306
jathena	2065	8410955
Pioneers	2024	953960
Qspacehulk	170	7258960
Qtads	1025	33202800
SLASH'EM	1642	33624200
Kategorie 3	**26745**	**349183598**
Bison	419	1890261
CompBenchmarks	268	185333
Concepts	108	35098
Jikes	2449	13236640
Make	531	41067

Fortsetzung

Name	RA	DA
SWIG	6848	3553943
tcc	277	398248
Kategorie 4	**10900**	**19340590**
AstroLib	15	581
Hunspell	509	356129
MUMmer	244	4197
muparser	20	12969
Open Dynamics Engine	6672	295993442
QUCS	3125	354536192
stardict	1220	19248
TextMatrix	310	74955192
units	82	15239
Kategorie 5	**12197**	**725893189**
AFPL Ghostscript	2439(*)	1995132143
Dia	2562(*)	5320528
Gimp	112674(*)	259080529
GLE	560	3449884
GNU Ghostscript	1626(*)	1249830650
GNU Plot	791	10189374
GQview	2876	792293
JLRFractal	18	44455
Pixie	1435	214000659
POV-Ray	444	6027832
Roadmap	1709	899164
RRDtool	192	295449
JOE	932	4562789

Fortsetzung auf der nächsten Seite...

Name	RA	DA
Sodipodi	4916(*)	15281679
trueprint	52	17982
xpdf	2565	24961485
Kategorie 6	**134859**	**3785324106**
AlsaPlayer	151	30838
Aqualung	1330	306816
SMPlayer	108	13767096
SoundTracker	721	119647
Streamripper	219	37477
transfig	278	1605598
Kategorie 7	**2807**	**15867472**
Clusterit	435	9260
FireStarter	599	134
GGMud	987	1688678
glBox	30	78
iometer	33	13244
Nagios	10305	4258340
SNNS	2180	457663
time	6	100
uucp	1693	4835014
zabbix	311	189887
Kategorie 8	**16579**	**11452398**
Bluefish	2129	51277
cscope	171	17002
Gwenhywfar	5552	25424060
OProfile	857	1393936

Fortsetzung

Name	RA	DA
nano	332	22534
sed	261	246499
Kategorie 9	**9377**	**30324161**
HT editor	1550	72719597
keyTouch	372	1935

Name	RA	DA
ScramDisk 4 Linux	35	247398
synergy	272	754186
Xplorer filemanager	91	3463241
Kategorie 10	**3177**	**78580293**

Werte mit (*) wurden mit einer einstufigen Richtungsanalyse ermittelt

Tabelle 5.3: Vergleich DAS mit Ergebnissen aus dieser Arbeit

Da eine erhebliche Diskrepanz zwischen den berechneten Werten von *function_pointer_check* und *das_tool* besteht, die sich kaum durch eine ungenaue Annäherung der Zeigeranalyse erklären lässt, wurde das kleine Testprogramm *fp.c* geschrieben.

```
#include <stdio.h>

struct x {int (*a) (int); int (*b) (int);};

int M1 (int v)
{
   int res;

   res = v*v;
   return res;
}

int M2 (int v)
{
   int res;

   res = v*v*v;
   return res;
}

int M3 (int v)
{
   int res;

   res = v*v*v*v;
   return res;
}

int main (void)
{
   struct x z;
   int (*fp) (int);
   int (*fp2) (int);
   int (*fs2) (int);
   int (*faa) (int);
   int (*st) (int);
   int (*fa[2]) (int);
   int res;

   fa[0] = &M1;
```

```
    fa[1] = &M2;
    z.a = &M2;
    fp = &M3;
    faa = fa[1];
    fp2 = fp;
    fs2 = z.a;
    st = fp2;
    res = fa[0](2) + fa[1](3) + z.a(2) -- fp(3);
    printf("%d", res);

    return 1;
}
```

Offensichtlich macht die durch *fp.c* berechnete Funktion keinen Sinn. Sie dient nur zur Überprüfung der Richtigkeit von *function_pointer_check*.

Während *function_pointer_check* ermittelt, dass es insgesamt 8 Ziele für Funktionszeiger gibt – fa[0], fa[1], z.a, fp, faa, fp2, fs2 und st haben jeweils ein Ziel – kommt *das_tool* auf 24! Da bereits bei diesem einfachen Programm ein recht großer Unterschied zwischen den beiden Ergebnissen auftritt – immerhin Faktor 3, erklärt dies die zum Teil gewaltigen Abweichungen bei den richtige Testprogrammen. Ein Vergleich der Werte mach aber keinen Sinn, da bei jedem beliebigen Programm in jeder beliebigen Kategorie festgestellt werden kann, dass die *DAS* Analyse eine viel zu große Zahl an potentiellen Zielen ausfindig macht.

5.3 Auswertung

Eine feldsensitive Zeigeranalyse würde bei dem Programmfragment

```
struct Feld {int *x, int *y};

int test (void)
{
    Feld *z;

    z->x = &a;
    ...
}
```

feststellen, dass nur die x-Komponente des Feldes z a als Ziel hat, nicht jedoch die y-Komponente. Eine feldinsensitive Analyse würde lediglich bemerken, dass dem Element z vom Typ *Feld* a als Ziel zugewiesen wird. Allerdings werden alle Felder verschmolzen und auch y bekommt bei dieser Analyse a als Ziel. Demnach wäre bei Elementen mit sehr vielen Feldern eine insensitive Analyse sehr ungenau und eine sensitive wäre wünschenswert. Allerdings werden lediglich bei der Zeigeranalyse nach *Wilson* Feld automatisch genau betrachtet. Dieses Verfahren ist aber mit einer exponentiellen Komplexität viel zu

teuer, um sie als Standard in jeden Compiler zu implementieren.

Möglicherweise würde es aber bereits genügen, diese nur bei Funktionszeigern zu verwenden. Daher wurde in dieser Arbeit das Vorkommen von Funktionszeigern in Strukturen gezählt, um eine Aussage über die Häufigkeit ihres Auftretens tätigen zu können. Analog dazu existiert auch für Arrays eine solche Analyse, bei der zwischen den einzelnen Elementen des Arrays unterschieden werden kann. Daher wurde ebenfalls die Häufigkeit von Funktionszeigern in Arrays bestimmt. Dabei stellte sich allerdings heraus, dass – zumindest bei den untersuchten Programmen – diese Verwendung von Funktionszeigern kategorieübergreifend praktisch nicht vorkommt. Sie stellt in keiner Kategorie mehr als 0,6% aller verwendeter Funktionszeiger dar. Daher ist eine teure arraysensitive Analyse in keinem Fall zu empfehlen, da die minimale Anzahl an gewonnenen Informationen den großen Aufwand in keiner Weise rechtfertigen würden.

Bei Feldern sieht dies allerdings ein wenig anders aus. Im Schnitt liegen 17% aller Funktionszeiger in Feldern. Teilt man die einzelnen Kategorie in 2 Gruppen, von denen die eine überdurchschnittlich viele Zeiger in Felder hält und die andere unterdurchschnittlich viele, so bilden die Kategorien 1 (21%), 3 (20%) und 8 (31%) die Gruppe mit vielen Verwendungen. Jedoch ist nicht nur die Anzahl der Funktionszeiger entscheidend, wenn es um die Wahl der richtigen Analysemethode geht. Da bei einer feldinsensitiven Analyse Ungenauigkeiten immer nur dann entstehen wenn sich das Ziel eines Zeigers ändert, spielt natürlich auch diese Änderungsrate eine gewichtige Rolle, um diese Entscheidung zu fällen. Daher sollte für jede Kategorie bestimmt werden wie viele Ziele einem Zeiger im Mittel zugewiesen werden und wie viel Ziele in jeder Rubrik im Durchschnitt maximal an einen Zeiger vergeben werden.

Name	FP in Structure	avg Ziele	max Ziele
Kategorie 1	**21%**	**1,1**	**4**
Kategorie 2	**13%**	**1,2**	**3**
Kategorie 3	**20%**	**1,2**	**4**
Kategorie 4	**8%**	**1,2**	**4**
Kategorie 5	**10%**	**1,1**	**5**
Kategorie 6	**16%**	**1,1**	**9**
Kategorie 7	**12%**	**1,1**	**3**
Kategorie 8	**31%**	**1,1**	**3**
Kategorie 9	**14%**	**1,2**	**7**
Kategorie 10	**16%**	**1,1**	**1**

Tabelle 5.4: Betrachtung der Ziele von Funktionszeiger

Bei dieser Betrachtung fällt auf, dass ein Großteil aller Funktionszeiger nur ein Ziel zugewiesen bekommen. Damit verbunden ist auch die Änderungsrate sehr gering. Somit wäre eine feldsensitive Analyse eher uninteressant, da die Menge der Ziele für die Funktionszeiger selbst bei einer May-Analyse, bei der einem Zeiger alle potentiellen Ziele zugeordnet werden, recht klein bleibt.

Dennoch erkennt man an den maximalen Zielen, dass es sehr wohl auch Funktionszeiger mit mehreren Zielen gibt. Gerade in den Bereichen Grafik und Software Development

existieren sehr viele Zuweisungen an einen Zeiger. Da sich in der Rubrik Grafik Funktionszeiger zu 16% in Strukturen befinden und sie damit nur knapp unterhalb des oben errechneten Durchschnitts liegt, könnte man sie dennoch zu den Kandidaten für eine feldsensitive Analyse zählen. Für Kategorie 10, die ebenfalls Funktionszeiger zu 16% in Strukturen unterbringt, sollte dieser Aufwand keines Falls betrieben werden, da hier eine neue Zuweisung an einen Funktionszeiger praktisch nicht stattfindet.

Insgesamt kommt diese Studie allerdings zu dem Ergebnis, dass sich ein Mehraufwand, den *Wilson* zu einem beinah linearen Verfahren wie *Steensgaard* darstellt, vermutlich nicht lohnen wird. Allerdings sei an dieser Stelle darauf hingewiesen, dass auch Verfahren wie *Steensgaard*, *Das* oder *Andersen* eine feldsensitive Analyse optional anbieten, was zwar ihre Komplexität erhöht, allerdings dennoch nicht an die von *Wilson* heranreicht. Daher könnte man bei Internetprogrammen, Spielen, Grafikprogrammen und Netzwerkprogrammen versuchsweise diese Option ziehen, um eventuell von den genaueren Ergebnissen zu profitieren.

6 (Kontroll-) Strukturen und Aufrufbeziehungen

6.1 Bearbeitung

Bevor auf das Analyseprogramm, das die Messwerte, die für diese Arbeit im Bereich der (Kontroll-) Strukturen und Aufrufbeziehungen von Interesse sind, eingegangen wird, sollten im Vorfeld zwei Begrifflichkeiten geklärt werden. So werden in dieser Arbeit all jene Funktionen als „reine Delegationen“ bezeichnet, die mindestens so viele Aufrufe wie Anweisungen besitzen. Diese Definition geht über die normale, in objektorientierten Sprachen gebräuchliche, Definition von einer Delegation hinaus. Dieses Beispiel soll die unterschiedlichen Modelle verdeutlichen:

```
class A {
   public:
      virtual void delegate() {
         printf("Delegation durchgeführt!");
      }
};
class B {
   public:
      A a;
      virtual void delegate() {
         a.delegate();
      }
};
```

Nach dem Verständnis einer OOP-Semantik wäre die Funktion *delegate* einer Instanz der Klasse B eine Delegation, da ein Aufruf *b.delegate* nur den Aufruf *a.delegate* durchführen würde. In dieser Arbeit gilt aber auch die Funktion *a.delegate* als Delegation, da die einzige Aufgabe die durch sie übernommen wird der Aufruf von *printf* ist. Somit findet das Analyseprogramm in diesem Codefragment zwei Delegationen, während man im klassischen Sinn nur eine erwarten würde.

Somit handelt es sich auch in diesem, um die Klasse C erweiterten, Beispiel bei der Methode *c.delegate* um eine Delegation, da die Anzahl der Aufrufe gleich der Anzahl der Anweisungen in der Methode sind:

```
class C {
   public:
      A a;
      B b;
      virtual void delegate() {
         a.delegate();
         b.delegate();
      }
};
```

Die Definition sagt aber nicht nur aus, dass die Anzahl der Aufrufe gleich der der Anweisungen sein muss, sondern, dass sie mindestens so groß sein muss. Dies ist wichtig, da sonst im Folgenden *e.delegate2* keine Delegation wäre, da in einer Anweisung zwei Aufrufe durchgeführt werden:

```
class D {
   public:
      virtual int delegate2(int x) {
         return x;
      }
};
class E {
   public:
      D d;
      virtual int delegate2(int z) {
         return d.delegate2(z) + d.delegate2(z);
      }
};
```

Dies hat aber auch zur Folge, dass *f.delegate2* ebenfalls als Delegation gilt, obwohl eine Anweisung, die keinen Aufruf beinhaltet in der Methode vorkommt:

```
class F {
   public:
      D d;
      virtual int delegate2(int z) {
         z = 42;
         return d.delegate2(z) + d.delegate2(z);
      }
};
```

Die andere Begrifflichkeit, die definiert werden sollte sind „Unterprogramme mit überproportional vielen Funktionsaufrufen“. Darunter werden all jeden Unterprogramme zusammengefasst, die verhältnismäßig viele Funktionsaufrufe – im Vergleich zu den anderen Unterprogrammen dieses IML Graphen – haben. So kann ein Unterprogramm in einem Testprogramm bereits mit einer sehr geringen Anzahl in diese Kategorie fallen, während eines mit deutlich mehr Aufrufen in einem anderen Testprogramm noch unter dem

Grenzwert liegt.

Nachdem nun erklärt wurde was mit *routine_check* berechnet wird, soll nun auf die Art und Weise wie dabei vorgegangen wird eingegangen werden. Zunächst müssen alle in einem Programm befindlichen Routinen gefunden werden. Da für jede dieser Routinen wichtige Informationen, wie die Anzahl der Verzweigungen, die Anzahl der Aufrufe und die Anzahl der vorkommenden Anweisungen gespeichert werden müssen, wurde der Datentyp *Routine_Record* eingeführt in dem neben diesen Werten auch Angaben wie Anfang und Ende im Quelltext oder der Name der Datei, in der der Quelltext steht, hinterlegt werden, um jede Routine eindeutig identifizieren zu können. Wurde im Graphen ein Element der Klasse *Routines* gefunden, so wird für die neue, nicht künstliche Routine ein solcher Record angelegt, die bereits bekannten Identifikationsmerkmale eingetragen und dieser in einer Liste gespeichert.

Nachdem nun alle in Programm vorkommenden Routinen erfasst wurden, müssen alle *Value* Knoten des IML Graphen besucht werden. In diesen Knoten befinden sich alle Elemente aus denen sich das Programm zusammensetzt, wie z.B. alle Variablen, Konstanten, Operatoren, ... An dieser Stelle sind aber nur nicht künstlich erzeugte Elemente aus den Bereichen *Conditionals*, *C_Switch_Statements* und *Try_Catch_Finally_Statements* von Interesse, da nach der Anzahl der Verzweigungen gesucht wird. Zwar kommt es auch bei allen Schleifen zu einer Verzweigung, jedoch liegt die Bedingung im Schleifenkopf in der Klasse *Conditionals* und somit werden auch diese mit den drei genannten Fällen abgehandelt.

Des weitern sollen auch alle Anweisungen gezählt werden. Als Anweisung gilt jede Zeile, die mit einem Semikolon abgeschlossen wird. Das Problem ist aber, dass es im Quelltext auch zu Sequenzen von Anweisungen kommen kann. So liegt z.B. bei

```
if (a == b)
{
    a =+ 1;
    b =- 1;
}
```

eine solche Sequenz aus insgesamt vier Anweisungen vor, da auch der Anfang der Sequenz ‚{‘ und das Ende ‚}‘ mitgezählt werden. Diese eingeschobenen Sequenzen und diese zusätzlichen Anweisungen sorgen dafür, dass Elemente aus der Klasse *Statement_Sequences* genauer untersucht werden müssen. Für jedes Element, das in einer Sequenz liegt und nicht künstlich erzeugt wurde oder aus einer der Klassen *Begin_Of_Lifetimes* ({) oder *End_Of_Lifetimes* (}) stammt, wird die Funktion *Get_Selected*, auf die später näher eingegangen wird, aufgerufen.

Liegt das Element in keiner der oben genannten Klassen, so kann es auch noch ein Aufruf einer Routine sein. Demnach müsste es in der Klasse *Routine_Calls* liegen. Ist auch dies nicht der Fall, so ist diese Element für die Analyse nicht weiter von Interesse und muss nicht weiter behandelt werden.

Ansonsten wird spätesten jetzt die Funktion *Get_Selected* aufgerufen, es sei denn sie wurde schon für Anweisungssequenzen benutzt. Dieser Funktion wird die Liste mit allen Routinen, die Fundstelle des aktuell betrachteten Elements und dessen Art – also

Verzweigung, Anweisung oder Aufruf – übergeben. Die Funktion überprüft daraufhin in welcher Routine das übergebene Element liegt, indem für alle Routinen, deren Quellcode sich in der selben Datei, in der auch das Element steht, befindet, die Anfangs- und Endzeile betrachtet werden und geprüft wird, ob diese das Element umschließen. Bei einem Treffer wird der dem Element entsprechenden Zähler erhöht.

Nachdem nun alle Kennzahlen des Testprogramms ermittelt wurden, kann die abschließende Berechnung und die Ausgabe erfolgen. Diese Aufgabe übernimmt die Funktion *Output_ To_ File*. Der dafür angelegte Record Type *Result_ Record* sammelt alle für die Analyse wichtigen Daten, wie die Anzahl der Unterprogramme mit keiner / einer / vielen Verzweigungen oder die Anzahl der reinen Delegationen, für das gesamte Testprogramm. Die Ausgabe erfolgt als CSV Datei die von der Prozedur *Secure_ Open* geöffnet oder, falls die angegebene Datei nicht geöffnet werden kann, angelegt wird.

In einem ersten Schritt wird bestimmt wie viele der Unterprogramme nur eine oder keine Verzweigung haben. Dazu wird einfach die Liste alle Routinen durchlaufen und geprüft, ob der Zähler der Verzweigungen auf 0 oder 1 steht. Danach wird die Überprüfung, ob es sich bei der Routine um eine Delegation handelt, durchgeführt. Hierfür wird der Prozentsatz Aufrufe zu Anweisungen bestimmt. Da es auch theoretisch leere Funktionen geben kann, muss dies zuvor abgeprüft werden, da sonst eine Division durch 0 vorliegt. Liegt dieser Prozentsatz bei 1 – oder unter 0, was durch leere Funktionen ausgelöst wird – wurde eine weitere Delegation gefunden. Schließlich wird noch bestimmt, ob in dem Unterprogramm keine, eine oder viele Aufrufe durchgeführt werden. Dies geschieht analog zur Bestimmung der Anzahl der Verzweigungen. Während diesen drei Berechnungen wird auch die Gesamtzahl der Aufrufe und Anweisungen ermittelt, um später Unterprogramme mit überproportional vielen Aufrufen identifizieren zu können. Der Quotient Gesamtzahl der Aufrufe zu Anweisungen wird analog zu dem jeder einzelnen Funktion berechnet.

In einem zweiten Durchlauf durch die Liste der Routinen berechnet man nochmal wie viele Prozent der Anweisungen Aufrufe sind und vergleicht diesen Wert mit dem des gesamten Programms. Liegt der des Unterprogramms darüber, so ist dies eine Routine mit überproportional vielen Aufrufen. Handelt es sich bei der Routine jedoch um eine Delegation, so wird diese nicht dazu gerechnet, da selbstverständlich jedes Unterprogramm bei dem in jeder Anweisung mindestens ein Aufruf steckt überproportional vielen Aufrufe beinhaltet. Ansonsten würden diese Routinen doppelt erfasst werden.

Da diese Berechnung, bedingt durch die mit unter lange Suche in den Listen, sehr viel Zeit in Anspruch nimmt, wird immer, wenn ein bestimmter Punkt passiert wird, ein ‚*‘ ausgegeben, damit der User erkennt, dass die Berechnung noch durchgeführt wird und kein Programmabbruch vorliegt.

Die Laufzeit von *routine_ check* wird zum einen durch die Anzahl der Routinen und zum anderen durch die Anzahl der Statements in dem untersuchten Programm bestimmt, da für jede Anweisung geprüft wird in welcher Routine diese liegt. Alle Routinen liegen dabei in einer Liste, die im worst case komplett durchlaufen werden muss. Daher kann die Komplexität mit $O(m * n)$ beziffert werden.

{routine_ check: 350 LOC}

6.2 Analyseergebnisse

In der folgenden Tabelle werden die Ergebnisse aus der Analyse *routine_ check* wiedergegeben. Die Prozentsätze in den einzelnen Spalten setzen sich aus den Beziehungen Anzahl der Aufrufe zu Anzahl der Anweisungen im ganzen Programm (C/S), Anzahl der Subroutinen mit keiner oder nur einer Verzeigung zur Anzahl aller im Programm vorkommenden Routinen (LB/R), Anzahl der reinen Delegationen zur Anzahl alle im Programm vorkommenden Routinen (D/R), Anzahl der Routinen mit keinem oder nur einem Aufruf zur Anzahl aller Routinen (LC/R) und Anzahl der Routinen mit überproportional vielen Aufrufen zur Anzahl aller Routinen (MC/R) zusammen. Bei Programmen, die aus mehreren ausführbaren Einheiten bestehen, wird ein Gesamtwert gebildet, wobei größere Einheiten (mehr Anweisungen oder mehr Routinen) stärker ins Gewicht fallen, als eher kleine Einheiten. Dies gilt ebenfalls für die zusammengefassten Werte ganzer Kategorien.

Name	C/S	LB/R	D/R	LC/R	MC/R
apache HTTP Server	38%	51%	21%	39%	36%
aria2	63%	80%	52%	52%	4%
Cntlm	47%	49%	17%	39%	35%
CTorrent	52%	56%	28%	53%	15%
DarkHTTPD	62%	56%	43%	35%	19%
define	47%	50%	14%	50%	29%
easyhttpd	62%	78%	41%	48%	3%
Expat XML Parser	12%	59%	25%	61%	35%
gSOAP Toolkit	72%	69%	49%	45%	3%
HTML Tidy	49%	41%	21%	37%	33%
LamaHub	59%	72%	38%	44%	10%
Licq	71%	78%	38%	50%	6%
mpop	53%	47%	23%	35%	27%
msmtp	51%	45%	26%	31%	28%
MyServer	72%	91%	37%	79%	6%
udt	64%	75%	48%	45%	5%
Urlgfe	79%	66%	41%	28%	9%
w3m	38%	39%	22%	40%	33%
wget	49%	38%	20%	30%	36%
Kategorie 1	**63%**	**39%**	**6%**	**58%**	**21%**
BASH	45%	44%	24%	33%	34%
bc	41%	49%	23%	33%	31%
DAR – Disk ARchive	63%	71%	47%	45%	7%
encrypt	71%	79%	64%	57%	0%
grep	36%	51%	25%	40%	38%

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	C/S	LB/R	D/R	LC/R	MC/R
Radmind	39%	40%	20%	33%	36%
random Filesplitter	46%	96%	28%	93%	11%
screen	33%	22%	9%	29%	40%
tar	47%	52%	40%	43%	26%
TCSH	48%	45%	20%	34%	31%
UnZip	33%	37%	18%	35%	33%
Kategorie 2	**61%**	**65%**	**42%**	**43%**	**13%**
Alchera – Game Editor	59%	55%	28%	38%	18%
Codebreaker	127%	74%	43%	38%	0%
Corewars	47%	49%	38%	39%	18%
dopewars	91%	45%	39%	25%	1%
Freeciv	48%	44%	24%	36%	22%
glBSP	47%	47%	17%	33%	33%
GNU Go	37%	53%	11%	38%	38%
jathena	36%	38%	10%	27%	39%
Pioneers	64%	60%	36%	43%	11%
Qspacehulk	64%	85%	49%	58%	3%
Qtads	60%	80%	44%	55%	7%
SLASH'EM	54%	24%	19%	28%	28%
Kategorie 3	**56%**	**56%**	**30%**	**42%**	**19%**
Bison	35%	56%	29%	43%	30%
CompBenchmarks	64%	77%	48%	44%	8%
Concepts	37%	53%	13%	45%	33%
Jikes	62%	76%	56%	51%	7%
Make	47%	37%	24%	26%	30%
SWIG	82%	45%	32%	24%	17%
tcc	32%	50%	15%	36%	43%
Kategorie 4	**63%**	**69%**	**46%**	**44%**	**12%**
AstroLib	36%	75%	31%	62%	18%
Hunspell	42%	52%	16%	53%	19%
MUMmer	68%	78%	49%	49%	3%
muparser	70%	79%	48%	51%	2%
Open Dynamics Engine	49%	71%	35%	62%	13%
QUCS	84%	89%	44%	67%	3%
stardict	68%	82%	43%	61%	4%
TextMatrix	61%	71%	46%	42%	6%
units	50%	46%	25%	28%	30%
Kategorie 5	**57%**	**75%**	**40%**	**58%**	**9%**

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	C/S	LB/R	D/R	LC/R	MC/R
AFPL Ghostscript	25%	51%	14%	49%	39%
Dia	73%	66%	35%	28%	8%
Gimp	50%	63%	18%	20%	27%
GLE	65%	75%	44%	49%	7%
GNU Ghostscript	25%	53%	16%	49%	38%
GNU Plot	40%	51%	21%	35%	33%
GQview	76%	57%	36%	28%	7%
JLRFractal	61%	86%	45%	59%	4%
Pixie	30%	81%	56%	68%	14%
POV-Ray	37%	59%	18%	51%	26%
Roadmap	58%	79%	38%	56%	10%
RRDtool	43%	40%	17%	29%	40%
Sodipodi	55%	46%	26%	24%	20%
trueprint	34%	49%	19%	31%	34%
xpdf	52%	73%	30%	63%	12%
Kategorie 6	**51%**	**64%**	**23%**	**31%**	**24%**
AlsaPlayer	67%	76%	46%	53%	3%
Aqualung	92%	73%	42%	54%	1%
SMPlayer	113%	94%	56%	69%	0%
SoundTracker	74%	53%	30%	28%	7%
Streamripper	39%	60%	24%	48%	28%
transfig	46%	41%	20%	28%	28%
Kategorie 7	**74%**	**76%**	**44%**	**54%**	**5%**
Clusterit	48%	56%	25%	41%	28%
FireStarter	122%	61%	60%	16%	0%
GGMud	70%	52%	31%	24%	11%
glBox	58%	55%	14%	40%	38%
iometer	61%	70%	41%	44%	9%
Nagios	54%	29%	12%	32%	18%
SNNS	23%	45%	15%	53%	38%
time	49%	47%	33%	33%	7%
uucp	41%	46%	19%	37%	32%
zabbix	41%	36%	17%	29%	37%
Kategorie 8	**46%**	**44%**	**20%**	**42%**	**27%**
Bluefish	102%	60%	52%	28%	0%
cscope	40%	42%	17%	30%	38%
Gwenhywfar	50%	50%	32%	44%	33%
JOE	46%	40%	23%	36%	32%

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	C/S	LB/R	D/R	LC/R	MC/R
nano	58%	43%	25%	30%	20%
sed	38%	32%	16%	27%	39%
Kategorie 9	**55%**	**49%**	**33%**	**40%**	**29%**
HT editor	43%	72%	33%	56%	21%
keyTouch	56%	63%	32%	54%	17%
OProfile	67%	77%	51%	46%	4%
ScramDisk 4 Linux	51%	75%	42%	50%	11%
synergy	64%	76%	52%	47%	6%
Xplorer filemanager	68%	75%	47%	52%	4%
Kategorie 10	**65%**	**76%**	**47%**	**49%**	**8%**

Tabelle 6.1: Ergebnisse von routine_check

Da bei der Betrachtung der Ergebnisse vor allem die C/S-Werte für Codebreaker (127%), FireStarter (122%) und Bluefish (102%) auffallen – diese Werte würden bedeuten, dass in diesen drei Programmen mehr Aufrufe als Anweisungen vorhanden sind, was zwar durchaus möglich ist, wie in *e.delegate2* aus dem Bearbeitungsteil dieses Kapitels bereits erklärt wurde, aber dennoch etwas verwundert – wurden zur Überprüfung der Korrektheit von *routine_check* der Quellcode der Hauptroutine von *code_breaker.c* manuell untersucht. Beispielhaft sei ein kleiner Ausschnitt daraus aufgeführt:

```
int up (int x)
{
...
  gdk_color_alloc (gdk_colormap_get_system (), &color_black);
  gdk_color_alloc (gdk_colormap_get_system (), &color_white);
  gdk_color_alloc (gdk_colormap_get_system (), &color_red);
  gdk_color_alloc (gdk_colormap_get_system (), &color_blue);
  gdk_color_alloc (gdk_colormap_get_system (), &color_yellow);
  gdk_color_alloc (gdk_colormap_get_system (), &color_green);
  gdk_color_alloc (gdk_colormap_get_system (), &color_cyan);
  gdk_color_alloc (gdk_colormap_get_system (), &color_gray);
  style[DEFAULT] = get_default_style ();
  style[BLACK]   = create_new_style (color_black);
  style[WHITE]   = create_new_style (color_white);
  style[RED]     = create_new_style (color_red);
  style[BLUE]    = create_new_style (color_blue);
  style[YELLOW]  = create_new_style (color_yellow);
  style[GREEN]   = create_new_style (color_green);
  style[CYAN]    = create_new_style (color_cyan);
  style[GRAY]    = create_new_style (color_gray);
  main_window = gtk_window_new (GTK_WINDOW_TOPLEVEL);
  game_board.main_app_window = main_window;
```

```
  gtk_signal_connect (GTK_OBJECT (main_window), "destroy",
                      GTK_SIGNAL_FUNC (gtk_main_quit), "quit");
...
}
```

Bei alleine diesen 20 Anweisungen kommt es zu 27 Aufrufen, was einen C/S-Wert von 135% ergeben würde. Da das restliche Programm mit dem oben genannten Code vergleichbar aufgebaut ist, kann der errechnete Wert von 127% durchaus stimmen.

6.3 Auswertung

Entscheidet man sich dazu vom Compiler bei jeder Verzweigung den Code verdoppeln zu lassen, so erhält man bei einem Programm mit vielen Verzweigungen – und damit verbunden vielen Codeverdopplungen – nach der Compilierung einen (Zwischen-)Code, der sehr groß werden und damit spätere Analysen und selbstverständlich auch die Ausführung des compilierten Programms gewaltig verlangsamen kann. Andererseits führen viele Verschmelzungen z.B. bei der Single Static Assignment Form zu sehr großen Phi Knoten Mengen – also zu ungenaueren Ergebnissen – was bei nachfolgenden Arbeitsschritten wie der Registervergabe schlechte Ergebnisse zur Folge hat.

Die Entscheidung, ob es sich lohnt Codeabschnitte bei Verzweigungen zu verdoppeln, hängt stark damit zusammen, wie sehr Verzweigt das jeweilige Programm ist. Die Auswertungen von *routine_check* haben ergeben, dass kategorieübergreifend bei einer überweltigenden Anzahl der Testprogramm (ca. 66%) über 50% aller Unterprogramme keine oder nur eine Verzweigung besitzen. Betrachtet man die Kategorien genauer, so kann man 2 Gruppen ausmachen. In der ersten Gruppe liegen mit den Kategorie 2, 4, 5, 7 und 10 die Programme, bei denen eine überdurchschnittlich große Anzahl der Unterprogrammen keine oder eine Verzweigung besitzen. Die restlichen Kategorien, also 1, 3, 6, 8 und 9 liegt dieser Prozentsatz unter dem Mittel von 67%. Die Entscheidung danach zu fällen, ob ein Programm der ersten oder zweiten Gruppe zugehörig ist, wäre aber eine zu grobe Einteilung.

Ein Unterprogramm wie

```
int up (int x)
{
   if (x > 0)
      return up(x - 1);
   else
      return 1;
}
```

besitzt zwar nur eine Verzeigung, aber durch die – in diesem Fall rekursiven – Aufrufe würden bei Codeverdopplung und Inlining dennoch ein sehr großer Code entstehen. Daher wurde bei *routine_check* auch die Anzahl der Aufrufe in den Unterprogrammen berechnet. Betrachtet man zunächst die Anzahl der Unterprogramme mit überproportional vielen Aufrufen verglichen mit der Anzahl aller Unterprogramme, so ergeben sich

erneut 2 Gruppen: Gruppe 1 hat unterdurchschnittlich viele Unterprogramme (weniger als 16%) mit überproportional vielen Aufrufen. In dieser Gruppe liegen die Kategorien 2, 4, 5, 7 und 10. Überraschender Weise stimmt diese Einteilung auch mit der obigen nach der Anzahl der Verzweigungen überein.

Ebenfalls wurden die Unterprogramme mit sehr wenigen (also mit keinem oder nur einem) Aufrufen gezählt. Eine kategorieübergreifende Auswertung hat ergeben, dass durchschnittlich 43% aller Unterprogramme in diese Rubrik fallen. Teilt man auch hier die einzelnen Kategorien in 2 Gruppen (eine mit Werten unter und eine mit Werten über 43%) ein, so bilden die Kategorien 1, 2, 4, 5, 7 und 10 die Gruppe mit wenigen Aufrufen. Sieht man von den Programmen aus dem Bereich Internet ab, so ergibt sich die bereits bekannte Einteilung.

Somit kann man sagen, dass Subroutinen der Programme aus den Bereichen Kommandozeile, Compiler, Education, Multimedia und OS Tools sowohl sehr wenige Verzweigungen als auch relativ wenige Aufrufe besitzen und sich daher als Kandidaten für Codeverdopplung anbieten würden.

Neben der Codeverdopplung kann auch Inlining von Interesse sein, da der Aufwand, den der Aufruf eines Unterprogramms erzeugt, größer sein kann, als wenn der komplette Code des Unterprogramms eingefügt wird. Auch können durch Inlining weiter Optimierungen ermöglicht werden. Bei großen, d.h. umfangreichen, Unterprogrammen ist allerdings der Aufwand, der durch den vergrößerten Code entsteht, verglichen mit den durch Inlining erzeugten Einsparungen, so viel größer, dass er sich nicht lohnt. Besonders reine Delegationen wären mögliche Kandidaten zum Inlining, da diese lediglich aus Aufrufen bestehen und keine „richtigen“, komplexen Berechnungen durchführen.[1]

Betrachtet man zunächst lediglich die reinen Delegationen, so sind die Kategorien 2, 4, 5, 7 und 10 potentielle Kandidaten für Inlining, da diese überdurchschnittlich viele Unterprogramme, die lediglich Delegationen sind, besitzen. Jedoch wäre diese Betrachtungsweise zu ungenau, da ein Programm, das in etwa so aufgebaut ist

```
int del1 (int x)
{
   return del2 (x);
}
int del2 (int y)
{
   return del3 (y);
}
...
int delx (int z)
{
   return nodel1 (z);
}
```

[1] vgl. Cavazos and O'Boyle [2005]

```
int nodel1 (int a)
{
   Statement1;
   ...
   Statementx;
}
...
int nodely (int a)
{
   Statement1;
   ...
   Statementx;
}
```

zwar sehr viele Delegationen besitzt, aber die Funktionen *nodel1* bis *nodely* aus so vielen Anweisungen bestehen, dass sich Inlining eventuell doch nicht lohnt. Daher wurde zusätzlich die durchschnittliche Anzahl an Anweisungen je Unterprogramm bestimmt. Alle untersuchten Programme bestehen aus Unterprogrammen, die aus durchschnittlich 20,2 Anweisungen bestehen. Unter diesem Wert, also sinnvolle Kandidaten für Inlining, sind die Programme aus den Kategorien 5, 6, 7 und 9.

Vergleicht man diese 4 Kandidaten mit den 6 Kandidaten aus der Delegationsanalyse, so kann man zu dem Schluss kommen, dass es sinnvoll sein könnte bei einem Compiler für Programme aus den Bereichen Education und Multimedia automatisch Inlining durchzuführen, bei Kommandozeilen Programme, Compilern Grafikprogrammen, Software Development und OS Programmen dem Nutzer den Vorschlag zum Inlining unterbreiten, da diese entweder sehr viele Delegationen besitzen oder aus sehr kurzen Unterprogrammen bestehen und bei Programmen aus den Bereichen Internet, Spiel und Netzwerk Inlining besser komplett zu unterlassen oder nur nach einer genauen Kosten-Nutzen-Abschätzung im Einzelfall durchzuführen.

Leider sind die Kategorien alles andere als homogen und daher liegt mit z.B. *Make* ein Testprogramm bei dem 63% aller Unterprogramm viele Verzweigungen besitzen und 74% viele Aufrufe durchführen auch Programme, die zur Codeverdopplung eher ungeeignet sind, in der Kategorie Compiler, für die diese Studie eine Empfehlung zu genau dieser Optimierung ausstellt. Selbige Ausreißer lassen sich auch für das Inlining finden.

7 Interprozedurale Analysen

7.1 Bearbeitung

Zur Bestimmung der gesuchten Kenngrößen des Aufrufgraphen wurde *callgraph_ check* erstellt. Das Programm beginnt mit der Suche nach der Wurzel des Aurufgraphen. Die IML stellt hierfür bereits die Funktion *Get_ Call_ Graph_ Root* in der Klasse *Systems* zur Verfügung. Dies ist wichtig, um die Tiefe des Aufrufgraphen bestimmen zu können.

Zunächst wird aber die Tiefe des Kontrollflußgraphen ermittelt. Auch diese kann direkt aus der IML ausgelesen werden. In der Klasse *Basic_ Blocks* steht im Attribut *Level* der Level des jeweiligen Grundblocks. Somit genügt es alle Grundblöcke zu durchlaufen und in einer Variable den größten Wert, den Level annimmt, zu speichern.

Nun muss der Aufrufgraph aufgebaut werden. Dafür wurde im Programm der Record-Type *My_ Routine* angelegt, in dem eine aufsteigende, eindeutige Nummer, der Eingangs- bzw. Ausgangsgrad – also die Anzahl der Aufrufstellen und Aufrufe – zur späteren Reduktion des Graphen, der Level der Routine, den Routine-Knoten aus der IML und das Hilfsfeld *Kill* gespeichert werden kann. Für jede nicht künstlich erzeugte Routine wird ein solcher Record in einer Liste angelegt. Die Felder mit dem Eingangs- und Ausgangsgrad und der Kopie des IML-Knotens können dabei bereits bei allen Routinen aus der IML übernommen und ausgefüllt werden, während der Level nur für die Wurzel mit 1 belegt werden kann.

Da im Attribut *Callees* der Klasse *Routines* bzw. in *Callers* der Klasse *O_ Routines* auch alles künstlich angelegten Routinen stehen, müssen die Felder des Eingangs- und Ausgangsgrads in einem weiteren Schritt angepasst werden. Dafür werden all diese künstlichen Routinen durchlaufen und jeweils überprüft, ob sie eine der „richtigen“ Routinen aufrufen oder von einer dieser aufgerufen werden. Bei einem Treffer wird der entsprechende Grad der betroffenen Routine verringert.

Die in *callgraph_ check* definierte Funktion *Get_ Level* bestimmt in einer solchen Liste aus *My_ Routine* Records den Level jedes Listenelements und gleichzeitig den maximalen Level des Graphen. Die Idee hinter dieser Funktion ist, dass alle Elemente des Level x eindeutige durch die Elemente auf dem Level x - 1 gefunden werden können. Die Liste wird hierfür maximal Anzahl der Listenelemente + 1 mal durchlaufen. Maximal deshalb, da die Schleife abbrechen kann, wenn kein Element des Levels y gefunden wird, da auch keins von Level y + 1, y + 2, usw. gefunden werden kann. Im n-ten Schleifendurchlauf werden also die Elemente auf dem Level n gesucht und bei jedem Treffer wird in einer inneren Schleife die Liste erneut durchlaufen. Trifft man dabei auf ein inneres Element, das vom äußeren Element aufgerufen wird, so liegt dieses innere Element auf Level n + 1, es sei denn ihm wurde bereits – z.B. bedingt durch einen Zyklus – ein geringerer Level zugeteilt. Der Maximale Level des Graphen ergibt sich automatisch aus der Anzahl der

Durchläufe der äußeren Schleife.

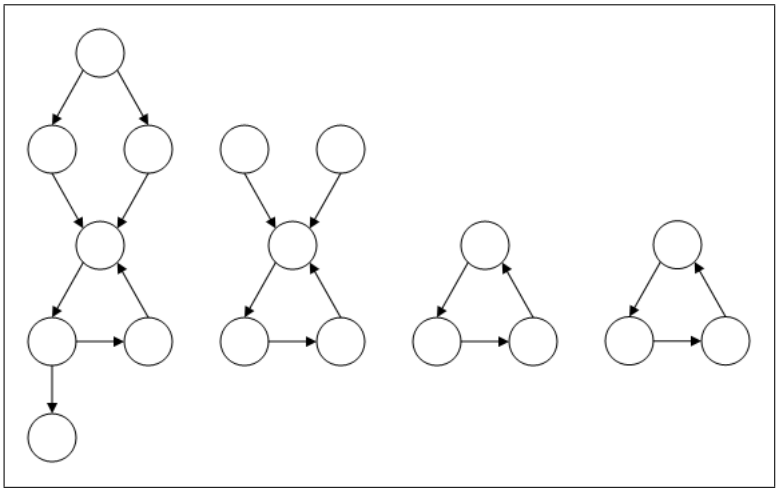

Abbildung 7.1: Ablauf der Reduktion eines Graphen

Bevor der Aufrufgraph nach Zyklen durchsucht wird, kann er zunächst – teilweise erheblich – reduziert werden. Dafür werden alle Routinen aus der Liste gestrichen, die einen Eingans- oder Ausgangsgrad = 0 haben, da diese definitiv nicht zu einem Zyklus gehören können. Die Funktion *Reduce_List* übernimmt diese Aufgabe. Dafür wird die Liste mit den Routinen so oft durchlaufen, bis keine weitere Reduktion durchgeführt werden kann. Wird in der Liste ein Element mit einem Grad = 0 gefunden, so wird dieses aber nicht gelöscht, sondern die bereits angesprochene Hilfsvariable *Kill* gesetzt, damit diese nicht mehr überprüft wird. Nun muss der *In_Degree* aller Listenelemente, die von dem zu löschenden Element aus aufgerufen werden, bzw. der *Out_Degree* von jenen, die diese Routine aufrufen, reduziert werden. Da es zu einer Veränderung kam, muss die Liste erneut durchsucht werden. Im schlimmsten Fall geschieht dies so oft wie es Routinen gibt.

Abbildung 1 zeigt wie eine solche Reduktion ablaufen kann. Schritt für Schritt werden alle Knoten mit Eingangs- oder Ausgangsgrad = 0 gelöscht. Im neu entstehenden Graphen werden daraufhin wieder für die übrigen Knoten der Grad bestimmt und alle auf die die Bedingung = 0 zutrifft werden eliminiert. Dies wird so lange durchgeführt, bis sich der Graph nicht mehr verändert.

Dass auf diese Weise ein zyklenfreier Graph vollständig reduziert wird, zeigt Abbildung 2, während Abbildung 3 zeigt, dass ein durch diese Reduktion entstehender Graph zwar mindestens einen Zyklus besitzt, dass dieser aber auch zyklenfreie Bereiche haben kann.

Dieser Idee hinter diesem Vorgehen beruht auf der Grundidee hinter A. B. Kahns Algorithmus zur Sortierung von großen Netzwerken.[1] Allerdings werden bei diesem Verfahren sowohl alle Knoten mit Eingangsgrad 0 und alle Knoten mit Ausgangsgrad 0 eliminiert, während Kahn lediglich einen Vergleichswert besitzt.

[1] vgl. Kahn [1962]

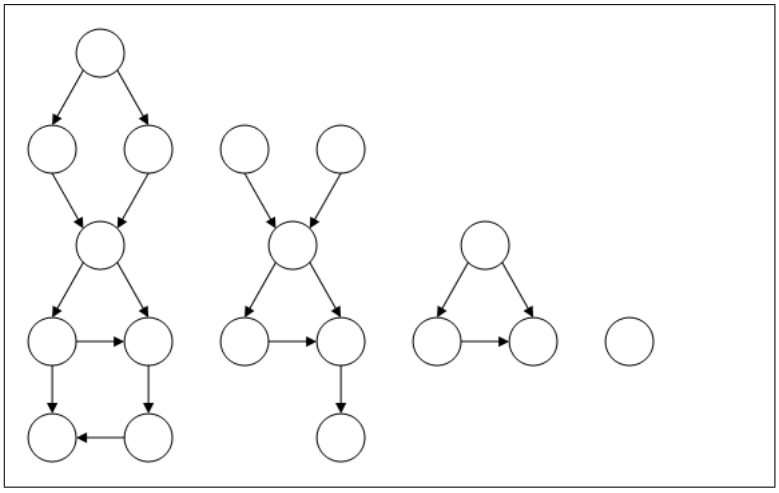

Abbildung 7.2: Vollständige Reduktion

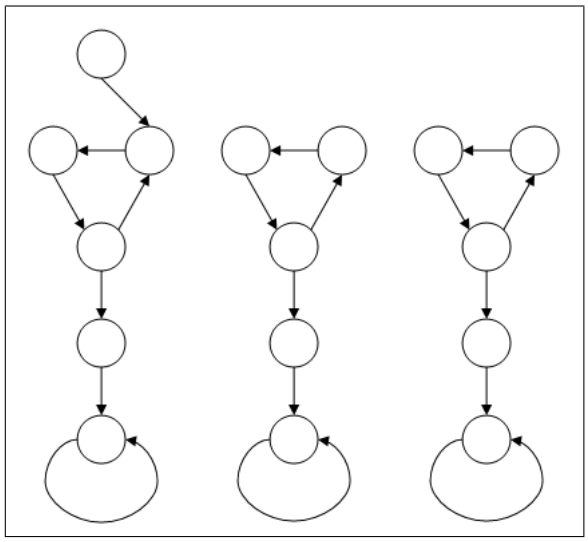

Abbildung 7.3: Reduktion mit zyklenfreien Bereichen

Bleiben nach dieser Reduktion noch Routinen übrig, bzw. wurde nicht bei allen Listenelementen *Kill* gesetzt, so gibt es definitiv mindestens einen Zyklus. Da aber auch die Größe, die genaue Anzahl der Zyklen und Informationen ob der Graph reduzibel ist von Interesse sind werden in *callgraph_check* die Record-Typen *Cycle_Help*, mit Angaben zur Entfernung zum möglichen Kopf des Zykluses, der Anzahl der Einstiegspunkte in den Zyklus und der Information, wer im Zyklus der Vorgängerknoten ist, und *My_Cycle*, in dem für jeden Knoten neben den bereits beschriebenen *My_Routine* Informationen eine Liste aus *Cycle_Help* Records existieren, eingeführt. Jeder *My_Routine* Knoten, bei dem nicht *Kill* gesetzt wurde, wird von *Reduce_List* abschließend zum *My_Cycle* Knoten erweitert und ein eindeutiger Name wird ihm zugewiesen.

Bei diesen Namen handelt es sich lediglich um eine Nummerierung der Knoten, um die Liste sehr leicht durchsuchen zu können und die Elemente schnell zu erkennen. Die Funktion *Get_Cycles* durchläuft dabei alle Knoten aus der Liste der potenziell zu einem Zyklus gehören Knoten und weist jedem Element die Entfernung 1 zu sich selbst zu. Dies wird so festgelegt, um aus der Entfernung direkt die Größe eines Zykluses ablesen zu können. Alle von diesem „Starknoten“ aus erreichbaren Knoten – also alle von dieser Routine aus aufgerufenen Routinen – bekommen die Entfernung 2. Alle Knoten die von Knoten mit der Entfernung 2 aus erreichbar sind bekommen die Entfernung 3 usw. Dies wird auf allen Pfaden so lange verfolgt, bis keine neuen Wege mehr gefunden werden – ruft sich z.B. die Routine mit der Entfernung x selbst auf, so bekommt sie selbstverständlich nicht die Entfernung x + 1! – oder die maximale Distanz (= Anzahl der Knoten + 1) erreicht wurde. Wird der Startknoten erneut erreicht, so wurde ein Zyklus gefunden. Hat einer der Knoten auf dem zum Zyklus gehörenden Weg einen Eingangsgrad > 1 und hat der Startknoten ebenfalls einen Eingangsgrad > 1, so ist der Aufrufgraph nicht reduzibel, da die Definition eines reduziblen Graphen aussagt, dass jeder Zyklus nur genau einen Eingang haben darf.

Grundsätzlich wird also der Algorithmus zur Breitensuche in Graphen von Edward F. Moore bei jedem Knoten angewandt, der eindeutig feststellt, ob ein Zyklus vorliegt oder nicht.[2] Allerdings wurden daran Anpassungen vorgenommen (z.B. Abstand eines Knotens zu sich selbst ist 1 und nicht 0), um die hier benötigten Informationen zu erhalten.

Das Problem bei dieser Methode ist lediglich, dass jeder Zyklus der Größe n n-mal gefunden wird – einmal mit jedem Knoten des Zykluses als Startknoten – was bei der Ausgabe des Ergebnisses korrigiert werden muss.

Da in *Get_Cycles* sehr viele Überprüfungen durchgeführt werden müssen, ist diese in die Unterfunktionen *Get_Distance* zur Berechnung der möglichen Pfade von jedem angegebene Knoten aus, *Distance* zur Bestimmung der aktuellen Entfernung eines Knotens, *TestReducible* zur Überprüfung, ob es auf dem aktuellen Pfad mehrere Eingänge gab, *Visited* damit sichergestellt werden kann, dass ein kleiner Zyklus in einem großen Zyklus keine Auswirkungen auf das Ergebnis hat, *SetReducible* und *Add_Cycle* zur Übergabe der Teilergebnisse in die Ergebnismenge und schließlich *Clear_Distance* damit alle veralteten Zwischenergebnisse gelöscht werden können, unterteilt.

Die Ausgabe erfolgt als *Comma Separated Values* (CSV), um eine möglichst einfache

[2] vgl. Moore [1959]

Einbindung in Excel erreichen zu können. *Output* und *Secure_ Open* übernehmen dies, wobei die Outputdatei erstellt wird, falls diese noch nicht existiert.

Show ist eine eher unbedeutende Hilfsfunktion, die alle Routinen, die in dem zu analysierenden Testprogramm vorkommen, mit ihrem Namen, dem jeweilen Eingangs- und Ausgangsgrad und dem bereinigten reduzierten Eingangs- und Ausgangsgrad ausgibt.

Die Komplexitäte von *callgraph_ check* ist mit $O(n^4)$ im worst case sehr hoch. n steht dabei für die Anzahl aller Routinen. Dieser hohe Wert wird durch die Funktion *Get_ Cycles* verursacht. Zunächst werden Zyklen der Länge 1 ... Anzahl der Routinen gesucht. Für jede Größe wird *Get_ Distance* aufgerufen, die ebenfalls eine Schleife Anzahl der Routinen mal durchläuft. In dieser Schleife wird eine Liste mit allen Routinen durchlaufen und bei jedem Element erneut die Liste abarbeitet. Durch die Funktion *Reduce_ List* werden allerdings viele Routinen aus diesen Listen gelöscht, so dass diese Komplexität im Mittel sehr stark verringert werden kann. Gerade bei Programmen, die Reduzibel sind oder gar keine Zyklen enthalten, schrumpft die Komplexität auf bis zu $O(n^2)$ zusammen.

{callgraph_ check: 639 LOC}

7.2 Analyseergebnisse

Bevor die unten stehenden Werte mit *callgraph_ check* berechnet wurden, wurde bei der jeweiligen IML zunächst mittels *das_ callgraph* und *iml2cg* die indirekten Aufrufe aufgelöst und diese daraufhin mit *iml2cfg* in die CFG Darstellung überführt.

Dies war aber bei einigen Programmen nicht möglich, da bei diesen – möglicherweise durch einen sehr großen Aufrufgraphen oder eine Endlosschleife ausgelößt – der Stack vollständig aufgebraucht wurde und *iml2cfg* eine Ausnahme wirft. In diesen Fällen wurden entweder lediglich die restlichen Dateien des Testprogramms untersucht oder, wenn keine Datei überführt werden konnte, in den Tabellen ein ‚k.a.‘ eingetragen.

Stellvertretend für alle Zyklen wird jeweils der mit den wenigsten und meisten Knoten genannt. Bei Testprogrammen, die mehrere IML Dateien erzeugen, werden bei den Angaben der Tiefe des Kontrollgraphen und des Aufrufgraphen sowie der Anzahl der Zyklen jeweils der minimale und maximale Wert genannt.

Um trotz der vielen ermittelten Werte noch eine möglichst hohe Übersichtlichkeit zu gewähren, wurde die Tabelle in zwei Teile aufgeteilt. Im ersten Teil befinden sich die Informationen wie hoch der Anteil der Programme mit einen reduziblen Aufrufgraphen ist und die minimale und maximale Tiefe von den Kontrollflussgraphen und den Aufrufgraphen, während sich der zweite Teil mit der minimalen und maximalen Anzahl der Vorkommnisse von Zyklen und deren Länge beschäftigt.

Name	Reduzibel	Min CFG Tiefe	Max CFG Tiefe	Min CG Tiefe	Max CG Tiefe
apache HTTP Server	88%	6	33	2	25
aria2	100%	34	34	12	12
Cntlm	100%	4	4	25	25
CTorrent	0%	18	18	9	9
DarkHTTPD	100%	14	14	7	7
define	100%	9	9	3	3
easyhttpd	100%	7	7	5	5
Expat XML Parser	0%	63	63	7	7
gSOAP Toolkit	0%	127	227	8	10
HTML Tidy	0%	39	39	9	9
LamaHub	100%	11	11	8	8
Licq	0%	35	35	12	12
mpop	k.a.	k.a.	k.a.	k.a.	k.a.
msmtp	k.a.	k.a.	k.a.	k.a.	k.a.
MyServer	60%	5	35	2	25
udt	k.a.	k.a.	k.a.	k.a.	k.a.
Urlgfe	0%	21	21	4	7
w3m	83%	5	103	4	25
wget	0%	33	33	9	9
Kategorie 1	**63%**	**4**	**227**	**2**	**25**
BASH	80%	5	133	2	22
bc	50%	64	88	7	7
DAR – Disk ARchive (nur *todos* betrachtet)	100%	3	3	25	25
encrypt	100%	14	14	2	2

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Reduzibel	Min CFG Tiefe	Max CFG Tiefe	Min CG Tiefe	Max CG Tiefe
grep	0%	102	102	6	6
Radmind	100%	7	27	3	7
random Filesplitter	100%	5	5	2	2
screen	0%	177	177	7	7
tar	50%	103	129	8	10
TCSH	50%	14	45	2	15
UnZip	67%	14	33	5	11
Kategorie 2	**77%**	**3**	**177**	**2**	**25**
Alchera – Game Editor	50%	9	28	4	25
Codebreaker	100%	15	15	5	5
Corewars	100%	56	56	6	7
dopewars	0%	26	26	8	8
Freeciv	0%	36	117	12	26
glBSP	0%	19	19	9	9
GNU Go	60%	7	98	6	25
jathena	0%	34	309	8	10
Pioneers	33%	23	23	6	25
Qspacehulk	0%	29	29	7	7
Qtads	0%	147	147	11	11
SLASH'EM	50%	24	108	5	11
Kategorie 3	**34%**	**7**	**309**	**2**	**26**
Bison	0%	126	126	8	8
CompBenchmarks	50%	32	33	8	9
Concepts	100%	15	15	8	8
Jikes	0%	103	103	17	17
Make	0%	23	23	8	8

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Reduzibel	Min CFG Tiefe	Max CFG Tiefe	Min CG Tiefe	Max CG Tiefe
SWIG	0%	453	453	16	16
tcc	50%	2	100	22	25
Kategorie 4	**33%**	**2**	**453**	**8**	**25**
AstroLib	100%	15	15	8	8
Hunspell	100%	11	29	3	7
MUMmer	100%	15	33	2	7
muparser	100%	28	28	9	25
Open Dynamics Engine	4%	17	55	6	25
QUCS	78%	6	186	2	11
stardict	83%	27	58	3	12
TextMatrix	0%	31	31	4	11
units	0%	29	29	6	6
Kategorie 5	**49%**	**6**	**186**	**2**	**25**
AFPL Ghostscript (ohne *gs*)	100%	7	19	2	5
Dia	0%	35	35	12	12
Gimp	1%	9	94	3	15
GLE	60%	9	116	2	12
GNU Ghostscript (ohne *gs*)	100%	6	19	2	5
GNU Plot	50%	8	145	4	18
GQview	0%	57	57	13	13
JLRFractal	100%	14	14	5	5
Pixie	80%	21	2049	3	21
POV-Ray	0%	100	100	11	11
Roadmap	54%	2	18	2	25

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Reduzibel	Min CFG Tiefe	Max CFG Tiefe	Min CG Tiefe	Max CG Tiefe
RRDtool	100%	2	95	5	25
Sodipodi	50%	5	62	2	12
trueprint	100%	27	27	8	8
xpdf	0%	56	56	12	15
Kategorie 6	**14%**	**2**	**2049**	**2**	**25**
AlsaPlayer	100%	55	55	8	8
Aqualung	0%	112	112	7	7
SMPlayer	100%	120	120	14	14
SoundTracker	0%	45	45	9	9
Streamripper	100%	5	31	5	8
transfig	50%	34	41	3	9
Kategorie 7	**77%**	**5**	**120**	**3**	**14**
Clusterit	100%	5	56	3	5
FireStarter	100%	13	13	7	7
GGMud	0%	121	121	13	13
glBox	100%	7	7	7	7
iometer	100%	16	16	8	8
Nagios	7%	56	150	3	10
SNNS	94%	2	59	2	25
time	100%	27	27	3	3
uucp	25%	24	68	4	8
zabbix	100%	10	27	5	10
Kategorie 8	**61%**	**2**	**150**	**2**	**25**
Bluefish	100%	66	66	9	9
cscope	100%	78	78	7	7
Gwenhywfar	17%	7	65	6	13

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Reduzibel	Min CFG Tiefe	Max CFG Tiefe	Min CG Tiefe	Max CG Tiefe
JOE	50%	10	68	2	9
nano	0%	35	35	7	7
sed	0%	36	36	14	14
Kategorie 9	**33%**	**7**	**78**	**2**	**14**
HT editor	50%	9	52	4	11
keyTouch	80%	2	17	8	25
OProfile	88%	10	52	7	15
ScramDisk 4 Linux	100%	3	54	2	6
synergy	0%	33	33	13	14
Xplorer filemanager	50%	7	34	4	15
Kategorie 10	**80%**	**2**	**54**	**2**	**25**

Tabelle 7.1: Ergebnisse von callgraph_check – Teil 1

Name	Min # Zyklen	Max # Zyklen	Min Zyklus	Max Zyklus
apache HTTP Server	0	604	1	18
aria2	4	4	1	2
Cntlm	0	0	–	-
CTorrent	5	5	1	2
DarkHTTPD	0	0	–	-
define	0	0	–	-
easyhttpd	0	0	–	-
Expat XML Parser	2150	2150	1	46
gSOAP Toolkit	29	378	1	18
HTML Tidy	96	96	1	5
LamaHub	0	0	–	-
Licq	18	18	1	10
mpop	k.a.	k.a.	k.a.	k.a.
msmtp	k.a.	k.a.	k.a.	k.a.
MyServer	0	10	1	4
udt	k.a.	k.a.	k.a.	k.a.
Urlgfe	16	28	1	8
w3m	0	314	1	18
wget	7	7	1	7
Kategorie 1	**0**	**2150**	**1**	**46**
BASH	0	1424	1	21
bc	2	7	1	3
DAR – Disk ARchive (nur *todos* betrachtet)	0	0	–	-
encrypt	0	0	–	-

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min # Zyklen	Max # Zyklen	Min Zyklus	Max Zyklus
grep	9	9	1	4
Radmind	0	3	1	1
random Filesplitter	0	0	–	-
screen	799	799	1	25
tar	1	51	1	17
TCSH	1	459	1	23
UnZip	0	3	1	5
Kategorie 2	**0**	**1424**	**1**	**25**
Alchera – Game Editor	0	44	1	12
Codebreaker	0	0	–	-
Corewars	7	7	1	2
dopewars	54	54	2	14
Freeciv	119	929	1	35
glBSP	14	14	1	2
GNU Go	0	63	1	8
jathena	22	643	1	16
Pioneers	3	222	1	12
Qspacehulk	36	36	1	7
Qtads	222	222	1	22
SLASH'EM	0	1149	1	26
Kategorie 3	**0**	**1149**	**1**	**35**
Bison	2673	2673	1	56
CompBenchmarks	3	4	1	2
Concepts	1	1	1	1
Jikes	3793	3793	1	51
Make	32	32	1	14

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min # Zyklen	Max # Zyklen	Min Zyklus	Max Zyklus
SWIG	670	670	1	15
tcc	0	70	1	18
Kategorie 4	**0**	**3793**	**1**	**56**
AstroLib	1	1	1	1
Hunspell	0	8	1	1
MUMmer	0	4	1	1
muparser	0	0	–	-
Open Dynamics Engine	1	86	2	4
QUCS	0	76	1	6
stardict	2	5	1	2
TextMatrix	2	2	2	2
units	5	5	1	7
Kategorie 5	**0**	**86**	**1**	**7**
AFPL Ghostscript (ohne *gs*)	0	1	2	2
Dia	2410	2410	3	60
Gimp	0	4755	1	36
GLE	0	61	1	13
GNU Ghostscript (ohne *gs*)	0	1	2	2
GNU Plot	3	591	1	23
GQview	379	379	1	23
JLRFractal	0	0	–	-
Pixie	2	177	1	14
POV-Ray	194	194	1	15
Roadmap	0	25	1	7

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min # Zyklen	Max # Zyklen	Min Zyklus	Max Zyklus
RRDtool	0	1	3	3
Sodipodi	0	1095	1	40
trueprint	1	1	1	1
xpdf	380	384	1	12
Kategorie 6	**0**	**4755**	**1**	**60**
AlsaPlayer	3	3	1	1
Aqualung	10	10	1	7
SMPlayer	0	0	–	-
SoundTracker	4	4	1	2
Streamripper	0	3	1	1
transfig	0	168	1	14
Kategorie 7	**0**	**168**	**1**	**14**
Clusterit	0	0	–	-
FireStarter	0	0	–	-
GGMud	134	134	1	22
glBox	0	0	–	-
iometer	2	2	1	2
Nagios	0	35	1	7
SNNS	0	27	1	10
time	0	0	–	-
uucp	0	1410	1	33
zabbix	1	1	1	1
Kategorie 8	**0**	**1410**	**1**	**33**
Bluefish	7	7	1	1
cscope	6	6	1	2
Gwenhywfar	0	1082	1	29

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Min # Zyklen	Max # Zyklen	Min Zyklus	Max Zyklus
JOE	0	4401	1	41
nano	25	25	1	11
sed	14	14	1	4
Kategorie 9	**0**	**4401**	**1**	**41**
HT editor	0	777	1	21
keyTouch	0	4	1	3
OProfile	0	18	1	2
ScramDisk 4 Linux	0	0	–	-
synergy	87	102	1	3
Xplorer filemanager	0	70	1	18
Kategorie 10	**0**	**777**	**1**	**21**

Tabelle 7.2: Ergebnisse von callgraph_check – Teil 2

7.3 Auswertung

Im Allgemeinen existieren für Graphen, die azyklisch oder reduzibel sind, deutlich leistungsfähiger und schnellere Algorithmen, als für irreduzierbare Graphen.[3] Als Beispiel sei an dieser Stelle die iterative Bestimmung der Dominatoren erwähnt, die diese mit nur einer Iteration bestimmen kann, wenn der Graph nicht irreduzibel ist.[4] Wenn sich aus den durch diese Arbeit gesammelten Daten ergeben würde, dass eine große Anzahl der zu compilierenden Programme einen Aufrufgraph besitzen, der reduzibel ist, so könnte man sich verstärkt auf einfach Algorithmen zur Graphenanalyse konzentrieren und dabei in Kauf nehmen, dass diese bei den wenigen nicht reduzierbaren Graphen deutlich länger brauchen.

Bei Programmen aus den Bereichen Spiele, Compiler, Education, Grafik und Software Development besitzt die Mehrheit einen irreduziblen Aufrufgraph. Die restlichen fünf Rubriken eignen sich vermutlich für einen einfachen Algorithmus, da sich deren Aufrufgraphen reduzieren lassen. Die Kategorie der Internet Programme besitzt dabei allerdings eine Sonderstellung, da *mpop*, *msmtp* und *udt* – bedingt durch Übersetzungsproblemen – nicht in das benötigte *CGF*-Format überführen ließen und daher das Ergebnis für die Rubrik noch nach oben oder unten verändern könnten.

Der Umkehrschluss, dass diese Analyse für die Aufrufgraphen, die sich nicht reduzieren lassen, gilt jedeoch nicht unbedingt. Dieses Programmfragment

```
...
int v0 (void)
{
   return v1(1) + v2(1);
}
int v1 (int x)
{
   return v3(x);
}
int v2 (int y)
{
   return v3(y);
}
int v3 (int z)
{
   if (z > 0)
      return v3(z -- 1);
   else
      return v4 ();
}
...
```

[3] vgl. Hecht and Ullman [1974]
[4] vgl. Staiger [2006], S. 207

das nicht reduzibel ist und diesen Aufrufgraph besitzt

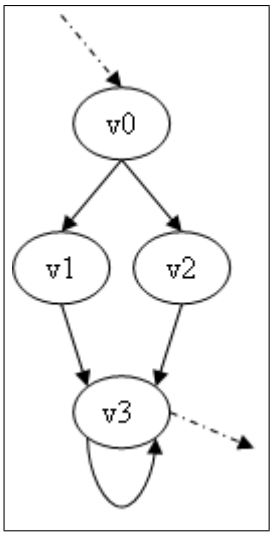

Abbildung 7.4: Aufrufgraph des Beispiels

könnte sich dennoch für eine einfache Analyse eignen, wenn der Graph vor *v0* und nach *v3* so weit zusammengefasst werden kann, bis er nur noch aus diesen 4 Knoten besteht.

Daher sollte in den stark irreduziblen Kategorien noch geprüft werden wie viele Zyklen in den irreduziblen Programmen enthalten sind und wie viele Knoten diese umfassen, da wenige oder sehr kleine Zyklen die Analysen nicht so sehr belasten und diese Graphen daher als beinahe reduzibel behandelt werden können.

In der folgenden Tabelle wurden daher in den fünf betroffenen Kategorien nur die irreduziblen Programme betrachtet. Dabei wird in jeder Kategorie die durchschnittliche Anzahl der Zyklen und die minimale und maximale Größe dieser Zyklen berechnet.

Name	Zyklen	minimale Zyklenlänge	maximale Zyklenlänge
Kategorie 3	**265**	**1**	**15**
Kategorie 4	**1207**	**1**	**34**
Kategorie 5	**67**	**2**	**4**
Kategorie 6	**92**	**1**	**30**
Kategorie 9	**719**	**1**	**21**

Tabelle 7.3: Analyse der irreduziblen Rubriken

Dank dieser neuen Zahlen sprechen jetzt auch Programme der Kategorie 5 dafür eine einfach Analyse für die Graphen zu verwenden, da durchschnittlich relativ wenige Zyklen in den Programmen vorkommen und diese maximal 4 Aufrufe umfassen. Außerdem sind beinahe 50% der Programme dieser Kategorie ohnehin reduzibel. Auch Kategorie 3 könnte für eine günstige Analyse in Frage kommen, da dort, obwohl recht viele Zyklen verwendet werden, diese nicht all zu groß sind.

Programme aus den Bereichen Compiler, Education und Software Development machen aber vermutlich wegen ihres komplexen Aufbaus, der nicht vereinfacht werden kann, Probleme und sollten daher eine eigene – speziell auf irreduzible Graphen angepasste – Analyse erhalten.

8 Zeiger- und Datenflussanalysen

Da die Messungen im Bereich der Zeiger- und Datenflussanalysen zwei komplett unterschiedliche Ziele verfolgen (NULL-Zeiger und Zeiger auf parallele Konstrukte), wurde dieses Kapitel in zwei separate Themenblöcke unterteilt.

8.1 Zeiger in Conditionals

8.1.1 Bearbeitung

Das Programm *cond_ check* dient zur Bestimmung der Häufigkeit von Zeigerüberprüfungen auf gleich bzw. ungleich NULL in Verzweigungen und setzt diese in eine Relation zu der Gesamtzahl der Verzweigungen mit Zeigern. Dazu benötigt man in einem ersten Schritt die Anzahl aller Verzweigungen. Jedwege Verzweigung – seien es case Statements, if Statements oder andere Bedingungen – werden in der IML in der Klasse *Conditionals* zusammengefasst. Auch Bedingungen z.B. aus einem Schleifenkopf oder Verknüpfungen von Bedingungen mittels *and* oder *or* werden als weitere if Statements interpretiert und daher ebenfalls bei *Conditionals* erfasst.

Somit reicht es aus, in einem Durchlauf alle *Conditional* Knoten des IML Graphen zu besuchen und bei jedem nicht künstlich erzeugten Knoten einen Zähler zu erhöhen. Der neue Record Type *Cond_ Count_ Record* wurde eingeführt, um alle Zwischenergebnisse in nur einer Variable speichern zu können und die Übergabe all dieser Ergebnisse an Unterfunktionen denkbar einfach zu gestalten.

Hat man einen natürlichen Conditional Knoten gefunden, so setzt direkt die Analyse der Bedingung ein, um darin enthaltene Zeiger zu entdecken und weitere Untersuchungen anstoßen zu können. Die Funktion *Check_ Cond* betrachtet die Bedingung dafür erst einmal hinsichtlich ihres logischen Operators. Da $<$ oder $>$ für Zeigervergleiche keine Rolle spielen, müssen lediglich die Conditions weiter analysiert werden, die auf Gleichheit oder Ungleichheit überprüfen, oder aus dem unären Operator *not* bestehen. Sollte der eigentlichen Bedingung ein *not* vorangestellt sein, so muss *Check_ Cond* erneut ohne die Negation, dafür mit dem Invers = True aufgerufen werden. Dies ermöglicht es eine Bedingung der Art

```
if !(p == NULL)
{
    SKIP;
}
```

nicht der Überprüfung gleich NULL, sonder ungleich NULL zuzurechnen, was die eigentliche Intention des Programms ist. Sollte die Bedingung in der Klasse *Equals* oder

Unequals liegen, so wird *Contains_Ref* angestoßen. Diese Unterfunktion ist zum einen dafür nötig, um Casts oder unäre Operationen auflösen zu können und hilft zum anderen dabei Bedingungen mit Zeigern zu finden, falls nicht wie üblich

```
if (p == NULL)
{
    SKIP;
}
```

sondern

```
if (NULL == p)
{
    SKIP;
}
```

benutzt wird. Dies geschieht dadurch, dass nicht die ganze Bedingung übergeben wird, sondern die linke und rechte Seite getrennt. Somit kann die Funktion sich selbst mit vertauschten Parametern nochmals aufrufen, falls der erste Durchlauf keinen Treffer brachte – d.h. es konnte keine Variable vom Pointertype auf der linken Seite der Bedingung gefunden werden.

Sollte sich auf der linken Seite ein Pointer befinden, so wird der Zähler für Bedingungen mit Pointer hochgezählt und gegebenen Falls Casts auf der rechten Seite aufgelöst. Steht nun auf der rechten Seite ein Element der Klasse Integer, so muss geprüft werden, ob es sich bei dem Element um *0* handelt. Eine spezielle Überprüfung auf den Typ *NULL* oder etwaige Makros der Art

```
#define NULLZEIGER (0)
...
if (p == NULLZEIGER)
{
    SKIP;
}
```

ist nicht nötig, da der Präprozessor diese Makros bereits vor der Übersetzung auflöst und der Compiler und somit auch die IML nur

```
if (p == 0)
```

sehen können. Tests für Funktionen mit dem konstanten Rückgabewert *0* könnten das Ergebnis zwar verändern, werden aber in *cond_check* nicht berücksichtigt, da eine solche Funktion sehr umfangreich sein kann, was es praktisch unmöglich macht den Rückgabewert im Voraus zu erkennen.

Liegt einer der auflösbaren *0*-Fälle vor, so wird der Zähler für gleich oder ungleich hochgezählt, während in allen anderen Fällen ein Zähler Rest angepasst wird.

Auch *cond_check* benutzt als Ausgabe eine CSV Datei in der neben den endgültigen Zählerständen auch weiter Berechnungen geschrieben werden. So ermittelt die Funktion *Output* die Relationen Anzahl der Conditions mit Zeigern = NULL zur Anzahl aller

Conditions mit Zeigern, Anzahl der Conditions mit Zeigern != NULL zur Anzahl aller Conditions mit Zeigern und Anzahl der restlichen Conditions mit Zeigern zur Anzahl aller Conditions mit Zeigern. Letzteres muss zwar nicht berechnet werden, da dies 100 - Relation 1 - Relation 2 ist, aber um Fehler während der Programmierung bereits frühzeitig erkennen zu können, wurde diese Berechnung dennoch in *cond_ check* aufgenommen, da eine Abweichung von den 100% um mehr als 1% nicht von einem Rundungsfehler, sondern eher von einem Programmierfehler herrührt. Auch können diese Relationen sehr einfach mittels Excel bestimmt werden, aber eine Berechnung ohne ein weiters Programm ermöglicht eine schnellere und direktere Bestimmung aller Werte von Interesse.

Die Komplexität von *cond_ check* liegt in $O(n)$, wobei n von der Anzahl der im Programm enthaltenen *Conditions*-Knoten und von der Anzahl der *Entity_ L_ Value*-Knoten abhängt. Alle Elemente beider Klassen werden in jeweils einer Schleife durchlaufen. Daher gilt $O(n)$ nicht nur im worst case, sondern für jedes Testprogramm.

{cond_ check: 249 LOC}

8.1.2 Analyseergebnisse

Die Ergebnisse der Analyse *cond_ check* werden in der folgenden Tabelle wiedergegeben. Wieder wurden Programme, die aus mehreren ausführbaren Einheiten bestehen, zusammengefasst und nur das aggregierte Ergebnis wird aufgeführt. Da das Ziel dieser Arbeit sein sollte, mögliche Programmiermuster in verschiedenen Rubriken zu finden, wird am Ende jeder Kategorie das Gesamtergebnis der inneliegenden Testprogramme berechnet.

Die einzelnen Spalten der Tabelle beinhalten den Namen des Programms, die darin enthaltenen Bedingungen, die Bedingungen mit Zeigervorkommnissen, die Anzahl der Bedingungen, die auf Gleichheit bzw. Ungleichheit mit NULL prüfen, die Anzahl der Zeigerverwendungen im Programm und schließlich der Anteil der Prüfungen auf NULL zu allen Conditions mit Zeigern.

Name	Cond	Cond mit Zeiger	= NULL	/= NULL	Zeiger	% NULL Cond
apache HTTP Server	4224	1020	157	174	14911	32%
aria2	160925	1661	8	329	266843	20%
Cntlm	439	103	16	0	1355	16%
CTorrent	857	9	2	1	1343	33%
DarkHTTPD	270	41	16	19	790	85%
define	44	3	3	0	69	100%
easyhttpd	2485	58	2	16	3333	31%
Expat XML Parser	2496	660	27	16	10365	6%
gSOAP Toolkit	23700	1111	8	59	58850	6%
HTML Tidy	2019	624	80	79	6956	25%

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Cond	Cond mit Zeiger	= NULL	/= NULL	Zeiger	% NULL Cond
LamaHub	17477	186	8	1	20412	5%
Licq	17416	935	276	324	44673	64%
mpop	17739	250	0	1	3934	0%
msmtp	17556	242	0	0	3303	0%
MyServer	3826	173	102	31	8778	77%
udt	58537	868	108	280	70909	45%
Urlgfe	1408	578	122	16	9276	24%
w3m	10103	2233	649	283	31242	42%
wget	3118	721	22	51	8527	10%
Kategorie 1	**344639**	**11476**	**1606**	**1680**	**565869**	**29%**
BASH	12855	2830	642	27	25207	24%
bc	1024	139	29	53	3703	59%
DAR – Disk ARchive	235259	8918	2275	3323	333964	63%
encrypt	42	18	15	3	189	100%
grep	1183	221	38	47	3063	38%
Radmind	3939	799	522	189	11146	89%
random Filesplitter	127	3	3	0	268	100%
screen	6384	1513	226	29	15343	17%
tar	3837	750	29	21	11202	7%
TCSH	5579	1206	286	269	13444	46%
UnZip	2766	187	70	66	2222	73%
Kategorie 2	**272995**	**16584**	**4135**	**4027**	**419751**	**49%**
Alchera – Game Editor	3013	632	149	368	8507	82%
Codebreaker	106	54	1	48	610	91%
Corewars	1944	490	10	113	5394	25%
dopewars	2358	603	4	10	9301	2%
Freeciv	62716	15267	335	1729	227542	14%
glBSP	852	261	8	1	2764	3%
GNU Go	15069	956	50	99	13504	16%
jathena	22566	3896	870	802	50655	43%
Pioneers	6496	1686	423	665	26669	65%
Qspacehulk	72245	1631	26	405	120027	26%
Qtads	48175	1795	276	675	87213	53%
SLASH'EM	53760	6893	39	545	57233	8%
Kategorie 3	**289300**	**34164**	**2191**	**5460**	**609419**	**22%**
Bison	2503	607	36	27	11615	10%

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Cond	Cond mit Zeiger	= NULL	/= NULL	Zeiger	% NULL Cond
CompBenchmarks	37997	327	0	12	55956	4%
Concepts	322	125	6	0	1646	5%
Jikes	58715	1097	22	16	70468	3%
Make	1877	678	229	365	6313	88%
SWIG	9930	1777	54	129	48641	10%
tcc	2117	173	6	13	4939	11%
Kategorie 4	**113461**	**4784**	**353**	**562**	**199578**	**19%**
AstroLib	205	12	2	9	367	92%
Hunspell	4038	125	27	49	8523	61%
MUMmer	23420	437	95	119	34785	49%
muparser	15485	206	4	42	27698	22%
Open Dynamics Engine	200126	7157	570	1526	688456	29%
QUCS	16082	1581	263	1067	44740	84%
stardict	20439	455	88	74	38720	36%
TextMatrix	136671	1805	316	324	208591	35%
units	460	185	3	0	1486	2%
Kategorie 5	**416926**	**11963**	**1368**	**3210**	**1053366**	**38%**
AFPL Ghostscript	50608	5840	1985	1151	152277	56%
Dia	4211	1965	223	618	31917	43%
Gimp	327394	80673	3128	21314	1345066	30%
GLE	126656	1954	217	518	187503	38%
GNU Ghostscript	40063	3983	1531	759	101499	57%
GNU Plot	9924	1097	106	207	18806	29%
GQview	5813	2230	10	19	3113	1%
JLRFractal	18215	396	1	96	28034	24%
Pixie	39644	3190	1540	872	254311	76%
POV-Ray	18369	846	352	398	18606	89%
Roadmap	35864	2698	870	1170	69401	76%
RRDtool	1758	225	79	20	5026	44%
Sodipodi	14889	6038	210	1558	64538	29%
trueprint	1031	122	32	11	1750	35%
xpdf	23254	104	0	0	46790	0%
Kategorie 6	**717693**	**111361**	**10284**	**28711**	**2356657**	**35%**
AlsaPlayer	3897	161	29	25	7598	34%
Aqualung	4147	683	192	307	16076	63%
SMPlayer	1949	3	1	0	3930	33%

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Cond	Cond mit Zeiger	= NULL	/= NULL	Zeiger	% NULL Cond
SoundTracker	3328	1061	115	718	11759	79%
Streamripper	2551	396	113	56	6015	43%
transfig	3800	616	141	148	12006	47%
Kategorie 7	**19672**	**2920**	**591**	**1254**	**57384**	**63%**
Clusterit	1652	494	151	283	4347	87%
FireStarter	657	170	61	45	4297	62%
GGMud	2796	672	75	37	15204	17%
glBox	95	39	13	7	287	51%
iometer	24122	174	19	10	24599	17%
Nagios	42815	16332	9663	3091	128252	78%
SNNS	29635	11620	2193	6990	100808	79%
time	44	12	3	1	225	33%
uucp	13259	3942	1461	1522	38070	76%
zabbix	3175	932	127	173	6716	32%
Kategorie 8	**118250**	**34387**	**13766**	**12159**	**322805**	**75%**
Bluefish	2706	1170	82	94	19934	15%
cscope	1382	236	64	70	3010	57%
Gwenhywfar	26949	12838	289	14	72096	2%
JOE	5972	1391	3	20	20158	2%
nano	1195	257	72	93	3595	64%
sed	1909	285	66	49	4846	40%
Kategorie 9	**40113**	**16171**	**576**	**340**	**123639**	**6%**
HT editor	9254	831	110	151	30916	31%
keyTouch	888	290	165	100	3160	91%
OProfile	239814	3779	53	567	366255	16%
ScramDisk 4 Linux	21108	237	19	65	29574	35%
synergy	73860	1856	106	668	118018	42%
Xplorer filemanager	107530	1368	40	283	190345	24%
Kategorie 10	**452454**	**8361**	**493**	**1834**	**738268**	**28%**

Tabelle 8.1: Ergebnisse von cond_check

8.1.3 Auswertung

Neben einer normalen Zeigeranalyse kann es sinnvoll sein eine präzise aber teure NULL-Zeiger-Analyse im Compiler zu implementieren, um frühzeitig sicher zu stellen, dass durch einen Zugriff auf NULL keine Ausnahmesituation eintritt. Eine so genau Analyse würde aber die Übersetzungszeit gewaltig verlängern. Daher wäre es wünschenswert, wenn unterschieden werden könnte, wann diese Analyse eingesetzt werden soll und wann

nicht, da z.B. durch den Programmierer bereits sichergestellt wurde, dass kein Zeiger auf NULL „falsch“ verwendet wird.

Bei der Auswertung der durch *cond_ check* erzeugten Ergebnisse fällt auf, dass sich die einzelnen Kategorien scheinbar sehr stark von einander unterscheiden. So wäre es sehr ratsam eine genaue NULL-Zeiger Analyse bei Programmen der Kategorie 9 durchzuführen, da bei diesen durchschnittlich nur 6% aller Conditionals in denen Zeiger verwendet werden auch überprüft wird, ob diese Zeiger auf NULL (bzw. nicht auf NULL) zeigen und so eine mögliche Fehlerquelle durch die Verwendung von NULL-Zeiger nicht ausgeschlossen wird. Betrachtet man gar das Verhältnis von Conditionals mit NULL-Überprüfung zu allen Conditionals, so machen diese nicht einmal 3% aus.

Ganz anders sieht es bei den Programmen aus der Rubrik „Netzwerk“ aus. Bei diesen werden immerhin in 22% aller Conditionals abgeprüft, ob ein Zeiger auf NULL zeigt. Beachtet man zusätzlich, dass Zeiger in Conditionals in dieser Kategorie insgesamt nur eine geringe Rolle spielen und man sich daher nur auf Conditionals mit Zeigern beschränkt, so erkennt man, dass diese Überprüfung ganze 75% ausmacht. Hier kann man sich aus Sicht des Compilers daher vermutlich darauf verlassen, dass der Programmierer sehr gewissenhaft alle kritischen Programmstellen mit einer solchen Abfrage bedacht hat und dass daher bei der statischen Analyse eine eher grobe Zeiger-Analyse ausreichen sollte.

Betrachtet man auf diese Art alle Kategorien, so lassen sich 3 Gruppen bilden. Kategorie 1, 3, 4, 9 und 10 prüfen in weniger als 30% aller Conditionals mit Zeigern darauf, ob es sich um einen NULL-Zeiger handelt. Daher wäre für diese Gruppe eine sehr genaue Zeigeranalyse ratsam. Allerdings deutet das Fehlen einer solchen Überprüfung nicht zwangsläufig auf schlampige und damit gefährliche Programmierung hin. Wird auf andere Weise sichergestellt, dass ein Zeiger nicht auf NULL zeigt – z.B. bei:

```
int main (void)
{
    int i;
    int j;
    int *p1;
    int *p2;

    i = 1;
    j = 2;
    p1 = &i;
    p2 = &j;

    if (*p1 != *p2)
        *p1 = 0;

    return 1;
}
```

ist eine zusätzliche Überprüfung vor der Zuweisung von *p1 unnötig, da dieser Zeiger definitiv auf i zeigt – spielt die Anzahl der Conditions mit NULL keine Rolle. Somit würde

für diese Funktion eine sehr einfach Zeigeranalyse ausreichen, ohne dabei an Sicherheit zu verlieren, obwohl hier sogar 0% aller Conditionals mit Zeiger – und damit auch 0% aller Conditionals – auf (Un-)Gleichheit zu NULL testen.

Die Kategorien 2, 5 und 6 liegen in einem Mittelfeld bei dem durchschnittlich 30% bis 50% aller Conditionals mit Zeigern auf NULL überprüft wird. Vergleicht man aber diese Überprüfungen mit der Anzahl aller Zeigerverwendungen, so stellt man fest, dass auf eine Überprüfung zwischen 51 und 230 andere Zeigerverwendungen auftreten. Da dieser Wert recht hoch ist, sollte auch bei dieser Gruppe an eine etwas genauere Zeigeranalyse gedacht werden. Selbstverständlich gilt auch hier die obige Einschränkung.

Die letzte Gruppe, die aus den Kategorien 7 und 8 besteht, sticht in beiden Werten heraus. So kommen hier nur 12 bis 30 Zeigeroperationen auf einen NULL-Test und verglichen zu allen Conditionals mit Zeigern machen diese Test 63% bis 75% aus. Dies spricht dafür, dass sicher programmiert wurde und eine spezielle NULL-Überprüfung im Compiler nicht mehr benötigt wird. Allerdings können wie bei

```
int main (void)
{
   int *p1;
   int *p2;

   if (*p1 != NULL)
      *p2 = 1;

   return 1;
}
```

diese Werte sehr hoch sein und dennoch kann p2 bei der Verwendung auf NULL zeigen. Somit sind all diese Werte keine Garantie dafür, dass sicher programmiert wurde, aber eine häufige Verwendung von NULL-Test spricht für eine vorsichtige und bedachte Verwendung von Zeigern.

Leider sind alle Kategorien ausgesprochen inhomogen. Kleinere Programme, bzw. Programme mit einer geringen Anzahl von Zeigerverwendungen oder Conditionals, werden dabei von großen Programmen aus der selben Kategorie unterdrückt. Daher wäre eventuell anzuraten im Compiler zwar eine spezielle Zeigeranalyse zu implementieren, diese aber nicht je nach Kategorie ein oder auszuschalten, sondern dem erfahrenen Benutzer erlauben diese auszuschalten, wenn er sich sicher ist, dass er alle gefährlichen Fälle einer NULL-Zeigerverwendung im Programm abprüft oder ausgeschlossen hat.

8.2 Zeiger auf parallele Strukturen

8.2.1 Bearbeitung

Um alle Vorkommnisse von Zeigern auf parallele Strukturen, also Threads, Semaphore, Condition Variables oder Mutex Locks in einem Programm finden zu können, wurde das Tool *pointer_check* erstellt. Dabei wird nach Variablen gesucht, die von einem der

folgenden Typen sind:

pthread_t: Datentyp zur eindeutigen Identifikation eines Threads. Unter Linux als einfacher Interger-Datentyp implementiert.

sem_t: Datentyp in C / C++ für Semaphore. Normalerweise als undurchsichtiger Record mit veränderbaren Feldnamen implementiert.

pthread_cond_t: Datentyp für Conditional Variables. In der Regel ebenfalls als opaker Record implementiert.

pthread_mutex_t: Datentyp für Mutex Locks. Auch hierfür wird ein opaker Record verwendet.

Diese Benennung der Variablentypen entsprechen der POSIX Norm IEEE 1003.1c von 1994[1]. Andere Datentypen, die die selben Aufgaben übernehmen, aber nach einem anderen Standard benannt sind oder gar vom Programmierer selbst erstellte Datentypen, werden von *pointer_check* nicht erkannt. Da jedoch alle „großen“ Betriebssysteme vollständig (Mac OS X oder viele Linuxdistributionen) oder zumindest weitgehend (Windows) POSIX-konform sind, sollte davon ausgegangen werden können, dass sich auch die Programmierer für diese Systeme an diesen Standard halten.

Das Programm *pointer_check* legt zunächst für jeden dieser 4 Typen eine Liste an, in der Elemente der IML Klasse *SLoc* gespeichert werden können. Diese Klasse stellt Informationen über die genaue Position eines Elements im Quelltext zur Verfügung. Diese Informationen reichen vom Dateinamen das Quellcodes, über die betroffene Zeile, bis hin zur konkreten Spalte in der sich das Element befindet.

Zusätzlich wird aber auch eine weitere Liste für jeden Datentyp der gesuchten parallelen Konstrukte angelegt, in dem Elemente der Klasse *O_Nodes* gespeichert werden können. Diese Listen werden benötigt, da nicht nach den Threads, Semaphore, Conditional Variables und Mutex Locks selbst, sondern nur nach Zeigern auf einen dieser Typen gesucht wird. Bevor diese Zeiger gefunden werden können, müssen aber erst einmal die Konstrukte selbst gefunden werden. Dies geschieht, indem einfach über alle *O_Nodes* iteriert wird und für jeden nicht künstlich erzeugten Knoten geprüft wird, ob sein *Unmangled_Name* – ein Attribut der Klasse *O_Node*, für dessen Bestimmung die Methode *Get_Unmangled_Name* zur Verfügung gestellt wird – einem der gesuchten Datentypen entspricht. Ist dies der Fall und das Element wurde noch nicht in einer der Listen aufgenommen, so wird es in die des gefundenen Datentyps eingetragen.

Wurden alle Variablen auf ihren Typ hin überprüft, kann das zu analysierende Programm erneut durchlaufen werden. Dieses mal werden alle nicht künstlichen *Value* Elemente vom Typ *T_Pointer* besucht. Zeigt einer dieser Zeiger auf ein Element der Klasse *T_User_Type* – wie es bei pthread_t, sem_t usw. der Fall ist – muss lediglich überprüft werden, ob die Deklaration dieses vom Programmierer angelegten Typs in einer der 4 *O_Nodes* Listen steht. Ist dies der Fall, so hat man einen Zeiger auf eine parallele Struktur gefunden. Hat man z.B. einen Treffer in der Liste der *pthread_mutex_t* und die

[1] vgl. IEEE [1995]

Position des Zeigers befindet sich noch nicht in der *SLoc* Liste der Mutex Locks, so wird dort diese Position eingetragen.

Wurden alle Zeiger überprüft, so befinden sich in den 4 Positionslisten alle Vorkommnisse von Zeiger auf die gesuchten parallelen Konstrukte. Was noch fehlt, ist die Ausgabe der Ergebnisse. Die Prozedur *Output* übernimmt dies und erzeugt je nach Listentyp – übergeben in dem Parameter *Kind* – eine angepasste Ausgabe in der neben der Information welche Konstrukte überhaupt verwendet wurden auch die genaue Stelle an der die Zeiger auftraten genannt wird.

Mit diesen Ergebnissen kann daraufhin von Hand – z.B. mit dem im Bauhaus Projekt enthaltenem *cobra* direkt auf der IML oder einem einfachen Texteditor wie *vim* auf dem ursprünglichen Quellcode – bestimmt werden, um welche Art der Verwendung bzw. um welche Programmiermuster es sich dabei handelt. In dieser Arbeit wird dabei zwischen einer „einfachen“ – im Sinne von „Eintritt in geschützten Bereich – Veränderung – Austritt“ – und einer komplexeren Variante unterschieden.

Von *pointer_ check* lässt sich die Komplexität mit $O(n^2)$ beziffern. n steht dabei für die Anzahl aller Variablenvorkommnisse aus dem untersuchten Programm. Diese werden alle einmal durchlaufen und für jeden Knoten, der auf einen Thread, ein Semaphor, eine Condition Variable oder ein Mutex Lock zeigt, wird geprüft, ob diese Stelle des Programms bereits in der Ergebnisliste aufgenommen wurde. Dieser worst case tritt nur ein, wenn alle Variablen des Programms Zeiger auf eine der 4 parallelen Konstrukte sind. Die Analyseergebnisse lassen allerdings vermuten, dass die zu erwartende Komplexität in der Praxis deutlich geringer ausfällt.

{pointer_ check: 205 LOC}

8.2.2 Analyseergebnisse

Die Analyse mit *pointer_ check* hat ergeben, dass ein Großteil der untersuchten Programme auf parallele Strukturen – und damit auch auf Zeiger, die auf selbige referenzieren – verzichten. Daher sollen in der folgenden Tabelle lediglich die Programme genannt werden, die auch Threads, Semaphore, Condition Variables oder Mutex Locks verwenden.

Name	Threads	Semaphore	Condition Variables	Locks
aria2	0	0	0	3
Cntlm	2	0	0	18
easyhttpd	0	0	0	6
gSOAP Toolkit	0	0	0	3
LamaHub	0	0	0	3
Licq	67	0	16	414
MyServer	2	4	7	33
udt	26	0	180	484

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Threads	Semaphore	Condition Variables	Locks
Kategorie 1	**97**	**4**	**203**	**964**
DAR – Disk ARchive	96	0	0	210
random Filesplitter	0	0	0	3
Kategorie 2	**96**	**0**	**0**	**213**
Qspacehulk	0	0	0	3
Qtads	0	0	0	3
Kategorie 3	**0**	**0**	**0**	**3**
CompBenchmarks	2	0	0	14
Jikes	0	0	0	3
Kategorie 4	**2**	**0**	**0**	**17**
MUMmer	0	0	0	27
muparser	0	0	0	6
Open Dynamics Engine	0	0	0	81
QUCS	0	0	0	21
stardict	0	0	0	18
TextMatrix	0	0	0	15
Kategorie 5	**0**	**0**	**0**	**168**
GLE	0	0	0	15
JLRFractal	0	0	0	3
Pixie	4	18	0	98
POV-Ray	0	0	0	3
Roadmap	0	0	0	6
Kategorie 6	**4**	**18**	**0**	**125**
AlsaPlayer	6	0	7	49
Aqualung	21	0	12	42
SMPlayer	0	0	0	3
SoundTracker	1	0	2	2
Streamripper	4	30	0	0
Kategorie 7	**32**	**30**	**21**	**96**
glBox	1	0	0	0
iometer	2	0	0	16
Nagios	2	0	0	10
Kategorie 8	**5**	**0**	**0**	**26**
—	—	—	—	—
Kategorie 9	**0**	**0**	**0**	**0**
HT editor	0	0	0	3
OProfile	0	0	0	18

Fortsetzung auf der nächsten Seite...

Fortsetzung

Name	Threads	Semaphore	Condition Variables	Locks
ScramDisk 4 Linux	0	0	0	12
synergy	22	0	10	16
Xplorer filemanager	0	0	0	6
Kategorie 10	**22**	**0**	**10**	**55**

Tabelle 8.2: Ergebnisse von pointer_check

8.2.3 Auswertung

Die Betrachtung der Zeiger auf parallele Strukturen scheint zu ergeben, dass gerade Mutual Exclusion Locks sehr häufig in Programmen verwendet werden. Schaut man aber auf die Herkunft diese Zeiger, so fällt auf, dass 330 dieser Zeiger auf Mutex Locks nur durch die Verwendung das Systemheaders *gthr-default.h* entstehen. In dieser Headerdatei, die ein Teil des GCC Pakets ist, werden Routinen zur Verfügung gestellt, die zu Threads kompatibel sind. Demnach würden diese Zeiger unter einer anderen Entwicklungsumgebung wegfallen und sollten daher an dieser Stelle nicht weiter untersucht werden. Diese Verwendungen machen alleine 20% aller Vorkommnisse von Zeigern auf Mutex Locks aus. Die restlichen sind allerdings auch nicht gleichmäßig verteilt. Nimmt man die drei Programme mit den meisten Zeigern auf parallele Konstrukte (*licq*, *udt* und *DAR – Disk ARchive*) aus der Wertung, so bleiben gerade einmal 424 Zeiger – oder anders ausgedrückt 19% der urprünglichen Zeiger – übrig.

Lässt man diese wenigen Ausreißer, die in unterschiedlichen Kategorien anzutreffen sind, außen vor, da diese nicht als repräsentativ angesehen werden können, so stellt sich die Frage, ob sich der Aufwand einer weiteren, teuren Zeigeranalyse überhaupt lohnt, zumal diese „parallelen“ Zeiger nicht einmal 0,06% an allen Zeigern ausmachen. Selbst mit der Headerdatei und den 3 Spitzenreitern stellen sie keine 0,3% aller Zeiger dar.

Dennoch soll kurz auf die Verwendungen der Zeiger eingegangen werden. Dabei werden die Verwendungen unterteilt in „normale“ Verwendungen von parallelen Konstrukten – also z.B. Konstrukt wird initialisiert, angefordert, Operation wird durchgeführt und wieder freigegeben – und in komplexere Konstrukte.

In den folgenden zwei Tabellen wird für jede Kategorie die Anzahl der Threads (T_x), der Semaphore (S_x), der Conditional Variables (C_x) und der Mutex Locks (M_x) in einfache (x_Nor) und komplexe (x_NN) Verwendungen unterteilt.

Name	T_Nor	T_NN	S_Nor	S_NN
Kategorie 1	**4**	**0**	**4**	**0**
Kategorie 2	**0**	**0**	**0**	**0**
Kategorie 3	**0**	**0**	**0**	**0**
Kategorie 4	**1**	**0**	**0**	**0**
Kategorie 5	**0**	**0**	**0**	**0**
Kategorie 6	**4**	**0**	**18**	**0**
Kategorie 7	**30**	**2**	**24**	**6**
Kategorie 8	**5**	**0**	**0**	**0**
Kategorie 9	**0**	**0**	**0**	**0**
Kategorie 10	**16**	**6**	**0**	**0**

Tabelle 8.3: Analyse der Zeiger auf parallele Konstrukte – Teil 1

Name	C_Nor	C_NN	M_Nor	M_NN
Kategorie 1	**7**	**0**	**32**	**4**
Kategorie 2	**0**	**0**	**0**	**0**
Kategorie 3	**0**	**0**	**0**	**0**
Kategorie 4	**0**	**0**	**8**	**0**
Kategorie 5	**0**	**0**	**0**	**0**
Kategorie 6	**0**	**0**	**66**	**32**
Kategorie 7	**19**	**2**	**56**	**34**
Kategorie 8	**0**	**0**	**6**	**17**
Kategorie 9	**0**	**0**	**0**	**0**
Kategorie 10	**10**	**0**	**10**	**0**

Tabelle 8.4: Analyse der Zeiger auf parallele Konstrukte – Teil 2

Kategorieübergreifend betrachtet sind gerade 12% aller Threads, 12% aller Semaphore, 5% aller Condition Variables und 33% aller Mutex Locks in komplexen Programmiermuster untergebracht. Nimmt man noch hinzu, dass Zeiger auf parallele Konstrukte ohnehin kaum Verwendung finden, kommt diese Arbeit zu dem Schluss, dass eine teure Zeigeranalyse, die extra auf solche Programmiermuster abzielt unnötig ist, da in der Regel, wenn im Programm überhaupt Parallelität vorkommt, lediglich ein Thread, ein Semaphor, ... generiert, benutzt und gleich wieder freigegeben wird. Für diese Fälle sollte eine normale Zeigeranalyse vollkommen ausreichen.

Sollte dennoch der Versuch durchgeführt werden, ob man durch eine stärkere Analyse Fehlerquellen besser eliminieren kann, so würden sich dafür Multimedia- oder Netzwerkprogramme eignen, da diese sowohl „normale“ als auch komplexe Verwendungsmustermuster für Zeiger aufweisen, wenn auch in einer recht geringen Zahl. Natürlich kann auch nicht ausgeschlossen werden, dass sich diese Statistiken sehr stark ändern werden, wenn immer mehr Hardware, die physikalische Parallelität unterstützt auf den Markt kommt, und somit der Nutzen, der durch diese Programmierweise erzielt werden kann, im Vergleich zur Sequenzialität erheblich gesteigert wird.

9 Abschlussbetrachtung

Diese Studie kam zu dem Ergebnis, dass für Funktionszeiger eine weitere, feldsensitive Analyse zum Einsatz kommen sollte, wenn das zu untersuchende Programm aus der Rubrik Internet, Spiele, Grafik oder Netzwerk stammt, auch wenn ein Großteil der Zeiger während ihrer Lebenszeit nur ein einziges Ziel zugewiesen bekommen. M. Hind und A. Pioli kamen bei ihrer empirischen Studie über verschiedene Zeigeranalysen allerdings zu dem Ergebnis, dass im Allgemeinen eine fluss-sensitive Zeigeranalyse den durch sie entstehenden Mehraufwand nicht rechtfertigen.[1] Daher sollte von *Wilson* abgesehen werden und lieber *Andersen* mit aktivierter Feld-Sensitivität verwendet werden.

Eine Optimierung bei der statischen Analyse durch Codeverdopplung kann bei den Rubriken Kommandozeile, Compiler, Lernsoftware, Multimedia und OS Tools erwartet werden, während sich Inlinig – laut den berechneten Ergebnissen – bei Lernsoftware und Multimedia bezahlt machen könnte. Bei dem Versuch einen Compiler für C Programme zu entwickeln, stießen R. Allen und S. Johnson bei ihrer Arbeit darauf, dass sich Inlining lediglich für kleine statische Funktionen lohnen wird.[2] Zu dem selben Schluss kam auch diese Studie. A. Ayers, R. Schooler und R. Gottlieb sind allerdings der Meinung, dass sich eine aggressive Strategie zur Codeverdopplung und Inlining bei jeder Funktion, in jedem Programm und in jeder Kategorie bezahlt machen wird.[3]

Bei den interprozeduralen Analysen kann in der Regel bei allen Programmen, die nicht aus der Kategorie Compiler, Lernsoftware und Software Entwicklung stammen, die gegebene Struktur des Aufrufgraphen beibehalten werden, während bei Programmen aus einer der drei genannten Rubriken eine Anpassung am Graphen durchgeführt werden sollte, da dieser für die normalen Analysen zu komplex ist. Eine solche Anpassung wurde von J. Janssen und H. Corporaal für Kontrollflussgraphen vorgestellt.[4] Dieses Verfahren beruht auf der Aufspaltung von Knoten, um so einen irreduziblen Graphen in einen reduziblen zu überführen. Danach können auf dem neuen Graphen bereits bestehende, optimierte Algorithmen angewandt werden, ohne dass die Verdopplung einzelner Knoten die Laufzeit erheblich einschränkt. Obwohl der Algorithmus für Kontrollflussgraphen entwickelt wurden, kann er dennoch für jeden beliebigen Graphen benutzt werden.

Bei der Zeigeranalyse kann die Empfehlung ausgesprochen werden, bei allen Rubriken außer bei Multimedia- oder Netzwerkprogrammen intensiv über eine zusätzliche Überprüfung auf NULL-Zeiger nachzudenken, da daraus möglicherweise resultierende Fehler vom Programmierer zu selten abgefangen werden. Zumal Algorithmen zu diesem Zweck existieren, die die anfallenden Compilierungskosten um lediglich 2,3% erhöhen, aber einen

[1] vgl. Hind and Pioli [2000]
[2] vgl. Allen and Johnson [1988]
[3] vgl. Ayers et al. [1997]
[4] vgl. Janssen and Corporaal [1997]

gewaltigen Vorteil bei der Ausführung – sowohl in Sachen Sicherheit, als auch bei der Geschwindigkeit – mit sich bringen.[5]

Dafür kann eigentlich bei allen Kategorien auf eine besondere Analyse für parallele Konstrukte verzichtet werden, da diese (noch) keine Rolle bei den untersuchten Programmen spielen. Das Fehlen von echten parallelen Programmen bestätigt auch der Technology Review von K. Greene.[6] Bei einem stärkten Einsatz von parallelen Konstrukten in den Programmen, sollten aber auch bessere Algorithmen bei der statischen Programmanalyse eingesetzt werden.

Aus den Ergebnissen, die bei den im Rahmen dieser Studie durchgeführten Analysen gewonnen wurden, kann der Eindruck gewonnen werden, dass man tatsächlich die statische Analyse anhand der Kategorien anpassen kann. Dies ist jedoch sehr mit Vorsicht zu genießen, da zum einen die Kategorien ausgesprochen heterogen sind und die Ergebnisse nur einen Mittelwert darstellen, von dem die einzelnen Programme zum Teil stark abweichen und zum anderen viele Programme in mehrere Kategorien passen und bei einer anderen Einteilung würde auch eine andere Empfehlung an die statischen Analysen ausgesprochen werden. In den meisten Fällen bringt – zumindest bei den hier gewählten Programmen – die Einteilung in die Kategorien keinen reellen Vorteil. Lediglich bei einigen Werten konnte eine ansatzweise Homogenität festgestellt werden. So treffen die ermittelten Einteilungen der Kategorien im Bereich der unterschiedlichen Ziele von Funktionszeiger auf die Mehrheit der inneliegenden Programme zu. Bei anderen Analysen, wie der Überprüfung auf NULL-Zeiger in Conditionals, existieren ebenfalls einige sehr homogene Kategorien (z.B. Compiler), bei denen sich zumindest eine Tendenz – sehr wenige oder sehr viele Überprüfungen – ausmachen lies. Die genannten Beispiele bilden aber leider eine Ausnahme.

Daher kam diese Studie zu dem Ergebnis, dass die verschiedenen Einstellungen für die statische Analyse nicht ausschließlich an der Kategorie festgemacht werden sollte, sondern es sollten sowohl einfach als auch komplexe Analysen in den Compiler implementiert werden, um dem (erfahrenen) Benutzer die Möglichkeit zu geben, von den für sein Programmiergebiet empfohlenen Einstellungen abzuweichen.

All diese Angaben beziehen sich allerdings auf die statische Programmanalyse, die in Compilern verwendet wird. Aber auch in Codeanalyseprogrammen, wie dem in dieser Studie verwendeten Bauhaus-Projekts, sind solche Programmanalysen im Einsatz. Für diesen Zweck wird natürlich eine andere Sichtweise auf die ermittelten Werte benötigt. Die Sicherheit der zu übersetzenden Programme spielt dabei eine sehr untergeordnete Rolle. Auch die Geschwindigkeit der Übersetzung ist weniger wichtig. Das Hauptaugenmerk bei solchen Codeanalyseprogrammen sollte auf der Genauigkeit der durchgeführten Analysen liegen, da anhand dieser Werte versucht werden soll, das Programm zu verstehen.

Bei der Funktionszeiger-Analyse sollte daher in allen Rubriken eine feldsensitive Analyse, z.B. nach *Andersen* oder sogar *Wilson*, zum Einsatz kommen, auch wenn die oben erwähnte Studie von M. Hind und A. Pioli einen zu geringen Nutzen darin sieht. Gerade beim Verglich der durch *function_pointer_check* bestimmten echten Ziele jedes Zeigers

[5] vgl. Kawahito et al. [2000]
[6] vgl. Greene

und den von *DAS* berechneten potentiellen Ziele, wurde eine solche Diskrepanz festgestellt, dass die auf diese Weise erhaltenen Werte kaum eine sinnvolle Aussage über das vorliegende Testprogramm abgeben. Daher sollte in diesem Anwendungsbereich die Genauigkeit der sehr viel höheren Laufzeit vorgezogen werden.

Auch wenn Codeverdopplung und Inlining den (Zwischen-)Code optimieren können, so verändern sie dennoch das originale Aussehen des Quellcodes und die darin vorkommenden Programmstrukturen. Gerade für Studien wie diese, die versuchen zu ermitteln, wie – d.h. mit welchen Programmiermustern – in der Praxis programmiert wird, ist dies nicht wünschenswert, da auf diese Weise Verfälschungen auftreten, die ausschließlich von dem verwendeten Codeanalyseprogrammen abhängen. Daher sollte mit solchen Optimierungen in diesem Bereich sehr vorsichtige umgegangen werden und auf keinen Fall sollte eine aggressive Strategie, wie die von A. Ayers, R. Schooler und R. Gottlieb, benutzt werden. Andererseits besteht natürlich auch ein sehr großes Interesse, dass Analysen, die später auf dem entstehenden (Zwischen-)Code arbeiten, möglichst genaue Ergebnisse erzielen. Aus diesem Grund wäre eine – wie bereits oben für die statische Analyse in Compilern angeführte – Strategie, die darin besteht, den Code bei Kommandozeilenprogrammen, Compilern, Lernsoftware, Multimediaprogrammen und OS Tools zu verdoppeln und bei Lernsoftware und Multimediaprogrammen Inlining zu bertreiben, anzustreben.

Betrachtet man die Aufrufgraphen, so wäre es – vor dem Hintergrund des Programmverstehens – ebenfalls wünschenswert, diese originalgetreu zu übernehmen, um die vom Programmierer modellierte Programmstruktur erkennen zu können. Wie oben bereits für die Compiler empfohlen, können diese auch für alle Programme, die keine Compiler, Lernsoftware oder Programme zur Software Entwicklung sind, beibehalten werden. Daher spricht sich diese Studie im Bereich der interprozeduralen Analysen in Codeanalyseprogrammen für die selbe Strategie wie bei oben bereits behandelten Compilern aus.

Im Bereich der Zeiger auf Conditionals muss die oben gegebene Empfehlung allerdings komplett umgekehrt werden. Im Bereich der Codeanalyse muss nicht auf Versäumnisse von Programmieren in Sachen Sicherheit bei Zeigerverwendungen eingegangen werden. Viel mehr sollte recht schnell ermittelt werden, wann z.B. in dem Codefragment

```
int rec (int *Zeiger)
{
...
   if (*Zeiger != NULL)
   {
...
      rec(Zeiger);
   }
   else
      return 1;
}
```

der Abbruch der Rekursion eintritt. Daher macht eine genaue Überprüfung auf NULL bei den Kategorien Sinn, in denen auch sehr viele Conditions mit = (bzw. !=) NULL verwendet werden. Bei Kommandozeilenprogrammen, Lernprogrammen, Grafikprogram-

men, Multimediaprogramme und Netzwerkprogramme sollte diese genauere Analyse deshalb verwendet werden. Kommt der oben erwähnte Algorithmus zum Einsatz, durch den ausgesprochen geringe Mehrkosten entstehen, so wäre eine solche Überprüfung in allen Kategorien – außer bei Programmen zur Software Entwicklung – ratsam.

Da die Werte bei der Analyse von Zeigern auf parallele Konstrukte keine statistische Relevanz haben, wird auch hier lediglich dazu geraten, zum jetzigen Zeitpunkt keine weiteren Analysen für diese Zeiger zu implementieren.

Auch wenn davon ausgegangen werden kann, dass Anwender, die Codeanalyseprogramme verwenden, das nötige Fachwissen haben, um ein Programm, das mehrere Argumente zur Auswahl der besten Analysestrategie erwartet, richtig bedienen zu können, sollte dennoch von dieser Möglichkeit abgesehen werden, da die Wahl der „richtigen“ Strategie einzig vom Aufbau des vorliegenden Testprogramms abhängt, dieses dem Nutzer eines Codeanalyseprogramms in der Regel aber vollkommen unbekannt ist. Aus diesem Grund kam diese Studie zu der Meinung, dass der starken Heterogenität der einzelnen Gruppen zum Trotz, die Wahl der Analysestrategie bei Codeanalyseprogrammen lediglich aus der Zuordnung des Testprogramms zu einer der zehn Kategorien zu erfolgen hat.

Anhang

Testrepository

Damit die erzielten Ergebnisse nachvollzogen werden können, sollen an dieser Stelle die Bezugsquellen des Quellcodes der Testprogramme genannt werden. Bei den Programmen *easyhttpd*, *Freeciv*, *SMPlayer* und *xpdf* war es leider nicht mehr möglich eine Quelle zu der analysierten Version anzugeben, da auf der offiziellen Programmseite nur noch eine aktualisierte Version angeboten wird.

Q1: `http://archive.apache.org/dist/httpd/`

Q2: `http://sourceforge.net/projects/aria2/`

Q3: `http://sourceforge.net/projects/cntlm/`

Q4: `http://sourceforge.net/projects/ctorrent/`

Q5: `http://dmr.ath.cx/net/darkhttpd/darkhttpd-1.2.tar.bz2`

Q6: `http://sourceforge.net/projects/define/`

Q7 (Neuere Version!): `http://sourceforge.net/projects/ehttpd/`

Q8: `http://sourceforge.net/projects/expat/`

Q9: `http://sourceforge.net/projects/gsoap2/`

Q10: `http://sourceforge.net/projects/tidy/`

Q11: `http://sourceforge.net/projects/lamahub/`

Q12: `http://sourceforge.net/projects/licq/`

Q13: `http://sourceforge.net/projects/mpop/`

Q14: `http://sourceforge.net/projects/msmtp/`

Q15: `http://sourceforge.net/projects/myserver/`

Q16: `http://sourceforge.net/projects/udt/`

Q17: `http://sourceforge.net/projects/urlget/`

Q18: `http://sourceforge.net/projects/w3m/`

Q19: http://ftp.gnu.org/gnu/wget/

Q20: http://ftp.gnu.org/gnu/bash/

Q21: http://ftp.gnu.org/gnu/bc/

Q22: http://sourceforge.net/projects/dar/

Q23: http://sourceforge.net/projects/encrypt/

Q24: http://ftp.gnu.org/gnu/grep/

Q25: http://sourceforge.net/projects/radmind/

Q26: http://sourceforge.net/projects/rand-file-split/

Q27: http://ftp.gnu.org/gnu/screen/

Q28: http://ftp.gnu.org/gnu/tar/

Q29: ftp://ftp.astron.com/pub/tcsh/

Q30: http://sourceforge.net/projects/infozip/

Q31: http://sourceforge.net/projects/mokoi/

Q32: http://hpux.connect.org.uk/ftp/hpux/Games/Board/codebreaker-1.2.1/

Q33: http://sourceforge.net/projects/corewars/

Q34: http://sourceforge.net/projects/dopewars/

Q35 (Neuere Version!): http://sourceforge.net/projects/freeciv/

Q36: http://sourceforge.net/projects/glbsp/

Q37: http://ftp.gnu.org/gnu/gnugo/

Q38: http://sourceforge.net/projects/jathena/

Q39: http://sourceforge.net/projects/pio/

Q40: http://r.vinot.free.fr/spacehulk/download.html

Q41: http://sourceforge.net/projects/qtads/

Q42: http://sourceforge.net/projects/slashem/

Q43: http://ftp.gnu.org/gnu/bison/

Q44: http://sourceforge.net/projects/compbench/

Q45: http://code.google.com/p/colibri-concepts/

Q46: http://sourceforge.net/projects/jikes/

Q47: http://ftp.gnu.org/pub/gnu/make/

Q48: http://sourceforge.net/projects/swig/

Q49: http://fabrice.bellard.free.fr/tcc/

Q50: http://mhuss.com/AstroCpp/AstroCpp.zip

Q51: http://sourceforge.net/projects/hunspell/

Q52: http://sourceforge.net/projects/mummer/

Q53: http://sourceforge.net/projects/muparser/

Q54: http://sourceforge.net/projects/opende/

Q55: http://sourceforge.net/projects/qucs/

Q56: http://sourceforge.net/projects/stardict/

Q57: http://sourceforge.net/projects/textmatrix/

Q58: http://ftp.gnu.org/pub/gnu/units/

Q59: http://sourceforge.net/projects/ghostscript/

Q60: http://ftp.gnome.org/pub/gnome/sources/dia/0.95/

Q61: ftp://ftp.gwdg.de/pub/misc/grafik/gimp/gimp/v2.2/

Q62: http://sourceforge.net/projects/glx/

Q63: ftp://ftp.dante.de/tex-archive/support/ghostscript/GPL/gs801/

Q64: ftp://ftp.gnuplot.info/pub/gnuplot/

Q65: http://sourceforge.net/projects/gqview/

Q66: http://sourceforge.net/projects/jlrfractal/

Q67: http://sourceforge.net/projects/pixie/

Q68: http://public.planetmirror.com/pub/povray/Official/Unix/

Q69: http://sourceforge.net/projects/roadmap/

Q70: http://oss.oetiker.ch/rrdtool/pub/

Q71: http://sourceforge.net/projects/sodipodi/

Q72: http://ftp.gnu.org/gnu/trueprint/

Q73 (Neuere Version!): `ftp://ftp.foolabs.com/pub/xpdf/`

Q74: `http://sourceforge.net/projects/alsaplayer/`

Q75: `http://sourceforge.net/projects/aqualung/`

Q76 (Neuere Version!): `http://sourceforge.net/projects/smplayer/`

Q77: `http://www.soundtracker.org/dl/v0.6/`

Q78: `http://sourceforge.net/projects/streamripper/`

Q79: `http://xfig.org/art15.html`

Q80: `http://sourceforge.net/projects/clusterit/`

Q81: `http://sourceforge.net/projects/firestarter/`

Q82: `http://sourceforge.net/projects/ggmud/`

Q83: `http://sourceforge.net/projects/glbox/`

Q84: `http://sourceforge.net/projects/iometer/`

Q85: `http://sourceforge.net/projects/nagios/`

Q86: `http://www-ra.informatik.uni-tuebingen.de/downloads/SNNS/`

Q87: `http://ftp.gnu.org/pub/gnu/time/`

Q88: `http://ftp.gnu.org/pub/gnu/uucp/`

Q89: `http://sourceforge.net/projects/zabbix/`

Q90: `http://sourceforge.net/projects/bluefish/`

Q91: `http://sourceforge.net/projects/cscope/`

Q92: `http://sourceforge.net/projects/gwenhywfar/`

Q93: `http://sourceforge.net/projects/joe-editor/`

Q94: `http://www.nano-editor.org/dist/v1.2/`

Q95: `http://ftp.gnu.org/gnu/sed/`

Q96: `http://sourceforge.net/projects/hte/`

Q97: `http://sourceforge.net/projects/keytouch/`

Q98: `http://sourceforge.net/projects/oprofile/`

Q99: `http://sourceforge.net/projects/sd4l/`

Q100: `http://sourceforge.net/projects/synergy2/`

Q101: `http://sourceforge.net/projects/cxplorer/`

Literaturverzeichnis

R. Allen and S. Johnson. Compiling c for vectorization, parallelization, and inline expansion. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 241–249, New York, NY, USA, 1988. ACM. ISBN 0-89791-269-1. doi: http://doi.acm.org/10.1145/53990.54014.

Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis.

Andrew Ayers, Richard Schooler, and Robert Gottlieb. Aggressive inlining. In *PLDI '97: Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 134–145, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: http://doi.acm.org/10.1145/258915.258928.

C. T. Bailey and W. L. Dingee. A software study using halstead metrics. In *Proceedings of the 1981 ACM workshop/symposium on Measurement and evaluation of software quality*, pages 189–197, New York, NY, USA, 1981. ACM. ISBN 0-89791-038-9. doi: http://doi.acm.org/10.1145/800003.807928.

Sören Berger. Analyse von Build-Prozessen. Studienarbeit am Institut für Softwaretechnologie der Universität Stuttgart, 2007.

John Cavazos and Michael F. P. O'Boyle. Automatic tuning of inlining heuristics. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 14, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 1-59593-061-2. doi: http://dx.doi.org/10.1109/SC.2005.14.

Manuvir Das. Unification-based pointer analysis with directional assignments. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 35–46, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: http://doi.acm.org/10.1145/349299.349309.

Kate Greene. Das Multi Core-Problem. `http://www.heise.de/tr/artikel/81305`.

M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21 (3):367–375, 1974. ISSN 0004-5411. doi: http://doi.acm.org/10.1145/321832.321835.

Michael Hind and Anthony Pioli. Which pointer analysis should i use? In *ISSTA '00: Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113–123, New York, NY, USA, 2000. ACM. ISBN 1-58113-266-2. doi: http://doi.acm.org/10.1145/347324.348916.

IEEE. *IEEE Std. 1003.1c-1995 thread extensions*. IEEE, 1995. ISBN 1-55937-375-X. Formerly POSIX.4a. now included in 1003.1-1996.

Johan Janssen and Henk Corporaal. Making graphs reducible with controlled node splitting. *ACM Trans. Program. Lang. Syst.*, 19(6):1031–1052, 1997. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/267959.269971.

A. B. Kahn. Topological sorting of large networks. *Commun. ACM*, 5(11):558–562, 1962. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/368996.369025.

Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. Effective null pointer check elimination utilizing hardware trap. *SIGPLAN Not.*, 35(11):139–149, 2000. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/356989.357002.

Rainer Koschke and Erhard Plödereder. Ansätze des Programmverstehens. In Franz Lehner, editor, *Softwarewartung und Reengineering*, pages 159 – 176. Deutscher Universitätsverlag, 1996.

Thomas J. McCabe. A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, page 407, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

Edward F. Moore. The shortest path through a maze. In *Proceedings of the International Symposium on the Theory of Switching*, pages 285–292. Harvard University Press, 1959.

Aoun Raza, Gunther Vogel, and Erhard Plödereder. Bauhaus - a tool suite for program analysis and reverse engineering. In Luís Miguel Pinho and Michael González Harbour, editors, *Reliable Software Technologies - Ada-Europe 2006*, volume 4006 of *Lecture Notes in Computer Science*, pages 71 – 82. Springer, Juni 2006.

Inbal Ronen, Nurit Dor, Sara Porat, and Yael Dubinsky. Combined static and dynamic analysis for inferring program dependencies using a pattern language. In *CASCON '06: Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 3, New York, NY, USA, 2006. ACM. doi: http://doi.acm.org/10.1145/1188966.1188970.

Stefan Staiger. Vorlesung Programmanalysen und Compilerbau. `http://www.iste.uni-stuttgart.de/ps/Lehre/WS0708/V_Programmanalysen/skript_pa_cb_ws06-online.pdf`, 2006.

Bjarne Steensgaard. Points-to analysis in almost linear time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, New York, NY, USA, 1996. ACM. ISBN 0-89791-769-3. doi: http://doi.acm.org/10.1145/237721.237727.

Robert P. Wilson and Monica S. Lam. Efficient context-sensitive pointer analysis for c programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1995. ACM. ISBN 0-89791-697-2. doi: http://doi.acm.org/10.1145/207110.207111.

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Unterschrift:

Auenwald, 04.12.2007