Institut für Softwaretechnologie
Abteilung Programmiersprachen und Übersetzerbau
Universität Stuttgart
Universitätsstraße 38
D – 70569 Stuttgart

Diplomarbeit

Empirische Bewertung verschiedener Ansätze der Datenflussanalyse

Empirical Evaluation of Variants of Data-Flow Analyses

Christian Nesich

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder
Betreuer:	Prof. Dr. rer. nat./Harvard Univ. Erhard Plödereder Dipl.-Inf. Torsten Görg

begonnen am:	18.07.2016
beendet am:	17.01.2017

CR-Klassifikation: D.3.4

Kurzfassung

Diese Diplomarbeit behandelt die Frage, ob es einen Ansatz gibt, der für alle Datenflussprobleme gleichermaßen geeignet ist und falls nicht, welche Ansätze sich für welches Datenflussproblem am besten eignen und warum.

Zu diesem Zweck werden verschiedene Datenflussprobleme ausgewählt, die dann mit unterschiedlichen Ansätzen empirisch miteinander verglichen werden.

Dafür wurde ein Interface implementiert, das es ermöglicht, unterschiedliche Ansätze einfach miteinander zu vergleichen.

Für die mit Hilfe dieses Interfaces implementierten Datenflussanalysen sollen nun, anhand repräsentativer Testprogramme, die Ausführungscharakteristika vermessen und beurteilt werden.

Abstract

This thesis addresses the question whether an approach exists that can be applied to all data flow problems and, if this is not the case, which approach would be the most suitable for a certain data flow problem and why.

Different data flow problems will be selected for this purpose, which will subsequently be compared empirically by applying different approaches.

An interface was implemented to this end which allows different approaches to be compared with each other.

The characteristic execution features of the data flow analyses, which were implemented with the assistance of this interface, shall now be gauged and assessed by means of representative test programs.

Inhaltsverzeichnis

1. Einführung 9
   1.1. Zielsetzung 9
   1.2. Überblick 9
2. Grundlagen 11
   2.1. Bauhaus 11
   2.2. IML 12
   2.3. Verwendete Datenflussanalysen 16
      2.3.1. Lebendige Variablen 16
      2.3.2. Gültige Definitionen 18
      2.3.3. Postdominanz 19
      2.3.4. Dominanz 19
   2.4. Verwendete Ansätze 20
      2.4.1. Traditionelle iterative Depth-first-Lösung 20
      2.4.2. Traditionelle iterative Lösung nach reversierter postfix-Ordnung 21
      2.4.3. Worklist-Ansatz, der nur die neu zu berechnenden Grundblöcke erfasst 22
      2.4.4. Worklist-Ansatz, der zusätzlich die reversierte postfix-Ordnung beachtet 23
3. Idee 25
   3.1. Bisheriger Zustand 25
   3.2. Framework in BAUHAUS 26
      3.2.1. Schnittstelle 27
      3.2.2. Interprozedurale Erweiterung 28
4. Implementierung 31
   4.1. Skeleton 31
      4.1.1. GDFA_Support 31
      4.1.2. GDFA_Traversals 32
      4.1.3. GDFA_Analysis 34
   4.2. Analysen 39
      4.2.1. Live_Variables 39
      4.2.2. Reaching_Defs 41
      4.2.3. Post_Dom 43
      4.2.4. Dom 44
5. Empirische Evaluation 45
   5.1. Testaufbau 45
      5.1.1. Korrektheit 45
      5.1.2. Metriken und Methodik 45
      5.1.3. Testsystem 47
      5.1.4. Testprogramme 47
   5.2. Messwerte 50

5.2.1. Laufzeit ... 50
5.2.2. Speicherverbrauch ... 52
5.2.3. Besuchsrate der Grundblöcke ... 54
5.2.4. Anzahl besuchter Grundblöcke ... 56
5.3. Vergleich ... 57
5.3.1. Intraprozedural – Gültige Definitionen ... 58
5.3.2. Intraprozedural – Lebendige Variablen ... 62
5.3.3. Intraprozedural – Postdominanz ... 67
5.3.4. Intraprozedural – Dominanz ... 71
5.3.5. Interprozedural – Gültige Definitionen ... 76
5.3.6. Interprozedural – Lebendige Variablen ... 80
5.4. Fazit ... 84
6. Zusammenfassung und Ausblick ... 85
6.1. Bewertung des Projekts ... 85
6.2. Ergebnisse ... 86
6.3. Erweiterungen und Verbesserungen ... 86
7. Literaturverzeichnis ... 87
8. Anhang A - Messwerte ... 89
8.1. Reaching Defs ... 90
8.1.1. Intra Procedural ... 90
8.1.2. Inter Procedural ... 91
8.2. Live Variables ... 92
8.2.1. Intra Procedural ... 92
8.2.2. Inter Procedural ... 93
8.3. Post_Dom ... 94
8.3.1. Intra Procedural ... 94
8.4. Dom ... 95
8.4.1. Intra Procedural ... 95
9. Erklärung ... 97

Abkürzungen

RPO	reverse Postorder
DFO	Depth-fist-Order
HPG	hierarchischer Programm Graph
IML	Intermediare Language

1. Einführung

Software ist in der heutigen Zeit aus kaum einem Bereich mehr wegzudenken. Damit kommt der Softwareentwicklung eine wichtige Bedeutung zu.

Im Prozess der Softwareentwicklung stellen Datenflussanalysen einen wichtigen Bestandteil dar. Bei Datenflussanalysen handelt es sich um eine grundlegende Technik, die von Compilern zur Codeoptimierung eingesetzt wird. Datenflussanalysen unterstützen aber auch den Entwickler beim Erkennen von Programmierfehlern.

Beide Anwendungsfelder profitieren davon wenn die Analysen möglichst schnell ausgeführt werden können. Je zeitkritischer der Anwendungsbereich ist, desto größer fällt auch der Vorteil aus. Am stärksten profitieren daher Techniken wie die Just-in-time-Kompilierung und Ansätze wie das Test-Driven-Development.

Es existieren bereits viele unterschiedliche Ansätze. Die bekanntesten Ansätze haben den Vorteil, dass sie theoretisch gut belegt sind. Unklar ist dabei jedoch noch, welche Ansätze sich für welches Problem am besten eignen.

Ein Vergleich unterschiedlicher Ansätze hinsichtlich ihrer Effizienz scheint daher vielversprechend.
Bereits [1] [2] liefern Hinweise darauf, dass die Wahl des Ansatzes einen großen Unterschied ausmachen kann. Die Autoren konnten gegenüber dem klassischen Round-Robin Algorithmus Geschwindigkeitsvorteile von 25% bis 40% aufzeigen.

Die hieraus resultierende Frage, welcher Ansatz denn nun für welche Datenflussprobleme optimal ist, soll Gegenstand dieser Arbeit sein.

1.1. Zielsetzung

Das Ziel dieser Diplomarbeit besteht darin, verschiedene Ansätze der Datenflussanalyse empirisch zu bewerten. Da unklar ist, welche Ansätze sich für welches Problem am besten eignen, sollen ausgewählte Ansätze empirisch miteinander verglichen werden.

Dabei soll die Frage geklärt werden ob es einen Ansatz gibt, der für alle Datenflussanalysen gleichermaßen geeignet ist. Falls es einen solchen Ansatz nicht gibt, soll geklärt werden, welcher Ansatz für welche Datenflussanalyse am besten geeignet ist.

Eine damit verbundene Aufgabe wird sein, ein Skeleton für Bauhaus zu entwerfen und zu implementieren, mit dem sich Datenflussanalysen so implementieren lassen, dass sich die zur Lösung verwendeten Ansätze mit minimalem Aufwand austauschen und vergleichen lassen.

Dieses Skeleton soll dann verwendet werden um die ausgewählten Datenflussanalysen mit den ausgewählten Ansätzen zu implementieren und so die Ansätze miteinander zu vergleichen.

1.2. Überblick

In Kapitel 2 werden die Grundlagen behandelt. Dies umfasst die Programmbibliothek Bauhaus inklusive der zugehörigen Zwischensprache IML, aber auch die in dieser Arbeit verwendeten Datenflussanalysen und Ansätze. In Kapitel 3 wird die Idee für die Umsetzung skizziert. Im Einzelnen wird dabei zuerst der bisherige Zustand beschrieben und dann die Idee vorgestellt. Anschließend wird die Implementierung in Kapitel 4 vorgestellt. Die Ergebnisse der Messungen werden in Kapitel 5 vorgestellt und diskutiert. Den Abschluss bildet Kapitel 6 mit einem zusammenfassenden Fazit.

2. Grundlagen

In diesem Kapitel werden die notwendigen Grundlagen vorgestellt, die für das weitere Verständnis der Arbeit notwendig sind. Im Rahmen dessen werden auch die Ergebnisse relevanter Publikationen zum Thema vorgestellt.

Direkt in Abschnitt 2.1 wird die verwendete Programmbibliothek Bauhaus vorgestellt. Im drauffolgenden Abschnitt 2.2 wird die zugehörige Zwischensprache IML skizziert. Den Abschluss des Kapitels bilden die Abschnitte 2.3 und 2.4, welche die verwendeten Datenflussanalysen und Ansätze beschreiben.

2.1. Bauhaus

Bei Bauhaus [3] handelt es sich um eine umfangreiche Toolsuite für Programmanalysen, die als Forschungsprojekt in Kooperation der Universitäten Stuttgart und Bremen entwickelt wird.

Teil von Bauhaus ist eine durch einen Compiler erzeugte Zwischensprache namens IML (Intermediate Language). Bauhaus ermöglicht auf der Basis von IML die Entwicklung vielerlei Analysen. Daher wird Bauhaus auch als das für diese Arbeit verwendete Datenflussframework dienen.

Die Analysekette von Bauhaus umfasst mehrere Schritte. Eine mögliche Analysekette ist in der Abbildung 2.1 zu sehen.

Abbildung 2.1: Analysekette

Mit jedem dieser Schritte wird die IML in zusätzlichen Schritten angereichert.

Im ersten Schritt werden die Quelldateien in IML umgewandelt. Die Umwandlung in IML erfolgt mit dem Tool cafeCC. Wenn es mehrere Quelldateien gibt, dann müssen die einzelnen IML-Dateien noch zu einer einzigen IML-Datei gelinkt werden. Das Linken der IML-Dateien geschieht mit dem Tool imllink.

Im zweiten Schritt ist es ratsam die Pointerziele zu ermitteln. Das Ermitteln der Pointerziele verbessert die Genauigkeit der nachfolgenden Analysen deutlich. Die Pointerziele können mit dem Tool pta_tool ermittelt werden.
Anschließend wird im dritten Schritt der Kontrollflussgraph, inklusive der Dominanzbäume, erzeugt. Der Kontrollflussgraph kann mit dem Tool iml2cfg erzeugt werden.

Das nächste und auch im Rahmen dieser Arbeit letzte wichtige Tool, bildet dann das thread_tool, welches parallele Programme analysiert. Es kann unter anderem zur Entdeckung von Data Races und Deadlocks eingesetzt werden.
Die im Rahmen dieser Arbeit erstellten Analysen erhalten eine IML-Datei als Eingabe, welche die Bauhaus Analysekette bis zum Thread-Tool durchlaufen hat.

2.2. IML

Bei der IML (Intermediate Language) handelt es sich um die von Bauhaus verwendete Zwischensprache. Im Rahmen der Studienarbeit [4] wurde eine Zwischendarstellung für Ada 83 entwickelt, welche später als Grundlage für die Entwicklung der IML diente. Viele der dort erarbeiteten Eigenschaften treffen auch heute noch auf die aktuelle IML-Version zu. Die Entwicklungsgeschichte der IML lässt sich folgendermaßen skizzieren.

1. Die erste wirkliche Version der IML entstand im Jahre 1998 im Rahmen der Studienarbeit [5], in welcher die Zwischensprache für die Sprache C erweitert wurde.
2. Drei Jahre später erfolgte durch die Diplomarbeit [6] die Erweiterung der IML für die Sprache Java.
3. Ein Jahr später kam mit der Diplomarbeit [7] noch die Sprache C++ dazu.
4. Die Diplomarbeit [8] erweiterte die IML im Jahre 2005 schließlich auch noch für Ada95.

Die IML ist ein mit Datenstrukturinformationen angereicherter Graph. Die Programminformationen werden in den Knoten gespeichert. Die IML wurde nach dem Vorbild des hierarchischen Klassenkonzeptes designet und richtet sich nach den in der Objektorientierung üblichen Notationen.

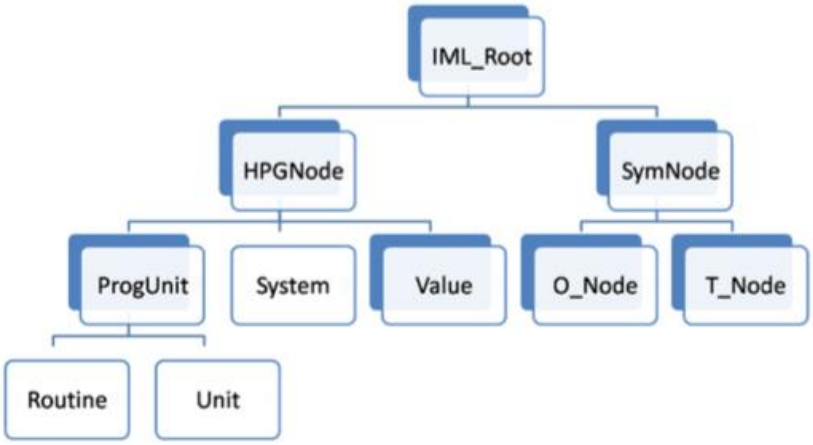

Abbildung 2.2.1: IML-Klassenhierarchie

Abbildung 2.2.1 zeigt, dass die IML-Klassenhierarchie im Wesentlichen aus abstrakten Klassen (in der Abbildung hinterlegt) besteht. Weiter sind die Anordnung des HPG (hierarchischer Programm Graph) und die semantischen Informationen in der IML-Klassenhierarchie erkennbar. Der HPG steht mit seiner Basisklasse HPGNode in der Klassenhierarchie direkt unter der Klasse IML_Root. Alle ausführbaren Teile des Programmes, d.h. alle Anweisungen und Unterprogramme, sind in Form eines abstrakten Syntaxbaumes im HPG enthalten.

Für die semantische Analyse zusätzlich benötigte Informationen werden normalerweise in einer Symboltabelle angelegt. Hier werden alle Bezeichner wie Variablen, Konstanten und Namen gespeichert. In der IML hat man hierfür eine andere Lösung gewählt. Hier wird keine Tabelle angelegt, sondern weitere Knoten mit der Basisklasse SymNode in den IML-Graphen mit aufgenommen. Der Bezug zu den Variablen und Unterprogrammen aus dem HPG wird über semantische Kanten auf Knoten vom Typ O_Node hergestellt. Die Knoten des IML-Graphen sind explizit als Objekte vorhanden, die Kanten existieren nur implizit in Form von Pointer Strukturen.

Weitere Schichten existieren oberhalb der Klasse IML_Root. In der Abbildung sind diese Schichten nicht vorhanden, da sie nur im Zusammenhang mit nicht generiertem Code eine Rolle spielen. Zu Gunsten der Übersichtlichkeit wurde auch auf die Darstellung der weiteren Klassen von HPGNode, die von Value abgeleitet werden, verzichtet.

Sehen wir uns nun die Knotentypen an, die für diese Arbeit wichtig sein werden.

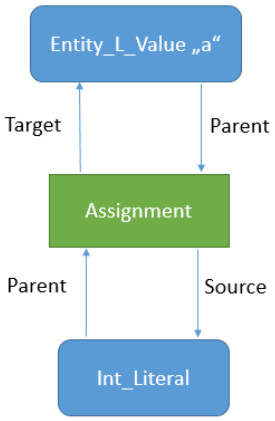

Abbildung 2.2.2: Modellierung von a=0

Die Abbildung 2.2.2 zeigt, wie eine einfache Zuweisung der Form a=0 in IML modelliert wird. Zuweisungen werden durch einen Assignment-Knoten modelliert. Ein Assignment-Knoten hat immer eine Quelle und ein Ziel. Ziel ist in diesem Fall die Variable a, die Quelle ist das Literal 0.

Variablen werden durch Entity_L_Value-Knoten modelliert, Literale durch einen ihrem Datentyp entsprechenden Literal-Knoten.
Jeder Entity_L_Value-Knoten verweist auf einen O_Variable-Knoten, über den sich feststellen lässt, ob zwei Entity_L_Value-Knoten zu derselben Variable gehören.

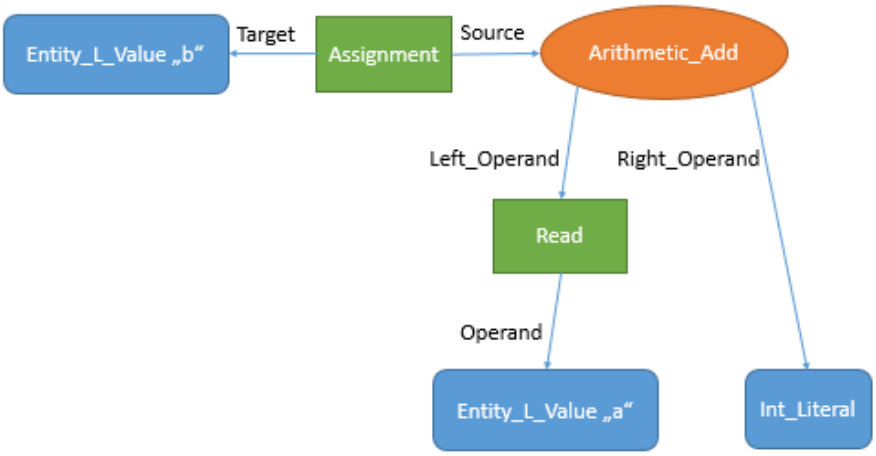

Abbildung 2.2.3: Modellierung von b=a+1

In Abbildung 2.2.3 sehen wir, wie eine Zuweisung der Form b=a+1 modelliert wird. Hier ist zu erkennen, dass ein Lesezugriff auf eine Variable über einen Read-Knoten erfolgt.
Ob eine Variable geschrieben oder gelesen wird lässt sich feststellen, indem man prüft, ob der entsprechende Entity_L_Value-Knoten einen Assignment- oder Read-Knoten als Parent hat.

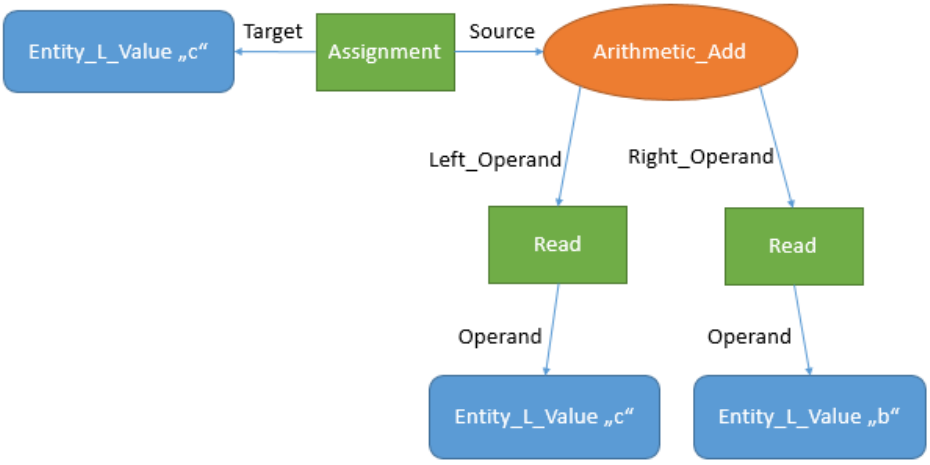

Abbildung 2.2.4: Modellierung von c=c+b

Wie eine Zuweisung der Form c=c+b modelliert wird, wird in Abbildung 2.2.4 dargestellt. Da hier zwei Variablen gelesen werden, sind in diesem Fall auch zwei Read-Knoten vorhanden. Die bisher besprochenen Knoten gehören fast ausschließlich zur Klasse der Value-Knoten. Eine Ausnahme bildet der O_Variable-Knoten der zur Klasse der O_Node-Knoten zählt.

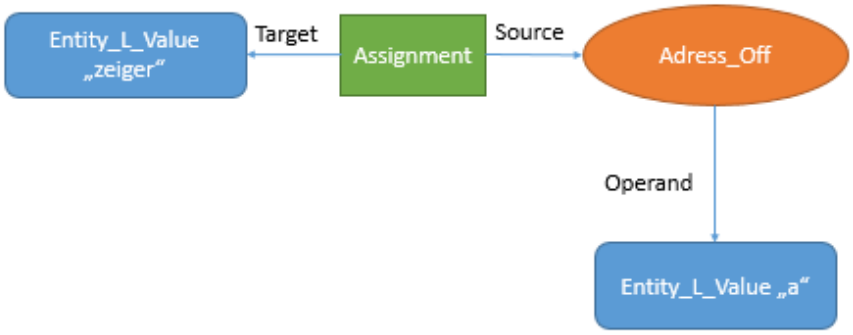

Abbildung 2.2.5: Modellierung von zeiger = &zahl

In Abbildung 2.2.5 ist zu erkennen, wie eine Zuweisung der Form zeiger = &zahl modelliert wird.
Eine Zuweisung an eine Pointer-Variable wird wie bei Variablen mit einem Assignment-Knoten modelliert. Quelle des Assignment-Knoten ist in diesem Fall ein Adress_Off-Knoten. Der Adress_Off-Knoten verweist auf das Ziel des Pointers.
Indem man prüft, ob der entsprechende Entity_L_Value-Knoten einen Assignment- oder Read-Knoten als Parent hat, lässt sich feststellen, ob eine Pointer-Variable geschrieben oder gelesen wird.

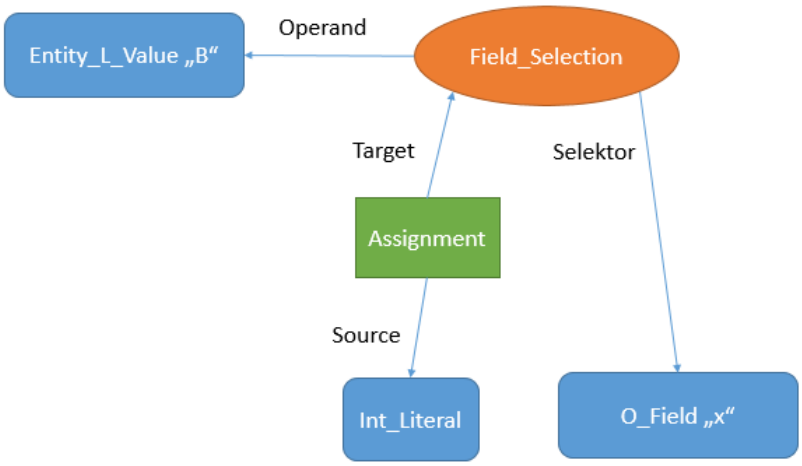

Abbildung 2.2.6: Modellierung von B.x = 7

Die Modellierung einer Zuweisung der Form B.x = 7 ist in Abbildung 2.2.6 zu sehen. Eine Selektion wird mit einem Field_Selection-Knoten modelliert. Bei einer Zuweisung an eine Selektion hat der Assignment-Knoten daher einen Field_Selection-Knoten als Ziel. Ob eine Selektion geschrieben oder gelesen wird lässt sich dadurch feststellen, indem man prüft, ob der Field_Selection-Knoten, den der entsprechende Entity_L_Value-Knoten als Parent hat, einen Assignment- oder Read-Knoten als Parent hat.

Der letzte noch zu erwähnende Knoten-Typ aus der Klasse der Value-Knoten ist der Direct_Call-Knoten, mit welchem der Aufruf einer Methode modelliert wird. Der Direct_Call-Knoten verweist dabei auf den O_Routine-Knoten der aufgerufenen Methode. Der O_Routine-Knoten verweist auf den zugehörigen Routine-Knoten.

Zu jedem Routine-Knoten gehört ein Kontrollflussgraph. Der Routine-Knoten selbst verweist dabei jeweils auf den Entry_BB- und Exit_BB-Knoten des Kontrollflussgraphen, von welchen aus dann über die übrigen Basic_Block-Knoten des Graphen traversiert werden kann.

Jeder Basic_Block-Knoten enthält eine Liste von Value-Knoten, durch welche die Statements im Basic-Block wie beschrieben modelliert werden. Die Reihenfolge, in der die Value-Knoten in der Liste auftauchen, ist die L-to-R Ausführungsreihenfolge. Diese ist wichtig, da nur darüber die Reihenfolge der Statements festgestellt werden kann.

Die Liste für das Beispiel aus Abbildung 2.2.3 sieht folgendermaßen aus:

1. 114 (Entity_L_Value „a“)
2. 103 (Read)
3. 104 (Int_Literal)
4. 94 (Arithmetic_Add)
5. 91 (Entity_L_Value „b“)
6. 83 (Assignment)

2.3. Verwendete Datenflussanalysen

Dieser Abschnitt stellt die verwendeten Datenflussanalysen vor. Die Auswahl der Datenflussprobleme fiel in Hinblick auf folgende Kriterien:

- alle ausgewählten Datenflussprobleme gehören mit zu den bekanntesten Datenflussproblemen
- die Datenflussprobleme wurden so ausgewählt, dass vergleichbare Probleme jeweils als Vorwärts- und Rückwärtsanalyse vorkommen

2.3.1. Lebendige Variablen

Die Analyse Lebendige Variablen [9] [10] [11] gehört mit zu den bekanntesten Datenflussanalysen.
Bei dieser Analyse wird ermittelt, ob eine Variable x an einem Punkt p lebendig ist. Die Variable x gilt am Punkt p als lebendig, wenn im Kontrollflussgraph ein Pfad von p bis zum Endknoten x existiert, auf dem der Wert von x verwendet wird.

Die Informationen, welche mit der Analyse Lebendige Variablen ermittelt werden, können unter anderem bei der Registervergabe verwendet werden. Stellt sich beispielsweise heraus, dass zwei Variablen nie gleichzeitig lebendig sind, so können sich diese problemlos ein Register teilen.
Die Datenflussgleichungen sehen folgendermaßen aus:

$$IN(EXIT) = BI$$

$$IN[B] = use \cup (OUT[B] - def)$$

$$OUT[B] = \bigcup_{S \text{ ist ein } Nachfolger \text{ von } B} IN[S]$$

Wie den Datenflussgleichungen zu entnehmen ist, handelt es sich um ein Rückwärtsproblem.

Die IN-Menge des EXIT-Knotens bildet die Menge BI. Im intraprozeduralen Fall ist BI die leere Menge. Im interprozeduralen Fall muss zwischen dem kontextsensitiven Fall und dem kontextinsensitiven Fall unterschieden werden.
Im kontextsensitiven Fall ist BI die Menge der Variablen, die an der entsprechenden Aufrufstelle lebendig sind. Im kontextinsensitiven Fall ist BI die Menge der lebendigen Variablen an allen zugehörigen Aufrufstellen.
Die OUT-Mengen und die IN-Mengen ergeben sich gemäß der obigen Mengengleichungen.
Die def-Menge enthält alle Variablen, die im Basic-Block B definiert werden. Eine Variable wird definiert, wenn eine Zuweisung an diese erfolgt.
Die use-Menge enthält alle Variablen, die im Basic-Block verwendet werden, bevor sie neu definiert werden. Eine Variable wird verwendet, wenn ein Lesezugriff auf diese erfolgt.

Unter der Annahme, dass die Variable x am Ende von Basic-Block B lebendig ist, ergeben sich folgende vier Möglichkeiten.

Fall	Lokale Informationen		Effekt auf Liveness
1	$x \notin Use\_B$	$x \notin Def\_B$	Liveness von x wird in B nicht beeinflusst
2	$x \in Use\_B$	$x \notin Def\_B$	Liveness von x wird in B generiert
3	$x \notin Use\_B$	$x \in Def\_B$	Liveness von x wird in B gekillt
4	$x \in Use\_B$	$x \in Def\_B$	Liveness von x wird in B nicht beeinflusst

Tabelle 2.3.1.1: Fallunterscheidung Liveness

Sowohl in Fall 1, als auch in Fall 2 und 4, ist Variable x am Eingang von Basic-Block B lebendig. Der Grund dafür ist jedoch für jeden Fall unterschiedlich.
Interessant ist hier vor allem der vierte Fall, da die Liveness von x in B gekillt, aber auch neu generiert wird. Fall 4 muss von Fall 1 und 2 unterschieden werden, da es Fälle gibt, in denen es wichtig ist zu wissen, ob x im Basic-Block B verändert wurde.

Wie [2] herausfanden, profitiert die Analyse Lebendige Variablen laufzeitmäßig davon, wenn möglichst wenige Basic-Blocks besucht werden. Es wurde festgestellt, dass für kleine Graphen eine Worklist-Implementierung mit „Priority Queue“ am besten abschneidet. Für große Graphen hingegen, soll eine Worklist-Implementierung mit Stack am schnellsten sein.

Folgende Tabelle zeigt deren Messergebnisse im Überblick:

Number of Blocks	Number of Graphs	Round Robin	Single Stack	Double Stack	Priority Queue	Simple Queue
Under 50	108	1	0.7748	0.7609	0.7449	0.7449
≥ 50 & < 100	35	1	0.6889	0.6914	0.6512	0.6812
≥ 100 & < 200	13	1	0.6091	0.5893	0.4828	0.5743
≥ 200 & < 400	7	1	0.8939	0.9072	0.5076	0.8575
≥ 400	6	1	0.4695	0.5124	0.4561	0.4773
10.000	1000	1	0.5863	0.5901	0.6034	0.5906
15.000	1000	1	0.5801	0.5845	0.5966	0.5859
20.000	1000	1	0.5636	0.5688	0.5799	0.5701
25.000	1000	1	0.5661	0.5718	0.5806	0.5736

Tabelle 2.3.1.2: Messergebnisse aus [2]

Die Zeiten sind in Sekunden angegeben.

Im Folgenden werden die Ansätze, die in Tabelle 2.3.12 vorkommen, kurz erklärt.
Bei den Ansätzen „Single Stack“, „Double Stack“, „Priority Queue“ und „Simple Queue“ handelt es sich um den Worklist-Ansatz, jeweils mit einem anderen Container für die Worklist. Das primäre Ziele der Autoren von [2] war, die unterschiedlichen Container für die Worklist zu vergleichen. Der Ansatz „Round Robin“ wurde nur zu Vergleichszwecken mit aufgenommen.

Der Ansatz „Round Robin“ iteriert dabei in reversierter Postfix-Reihenfolge über die Graphen. Als einfachsten Container für die Worklist wird ein einfacher Stack angegeben. Dieser Ansatz wird „Single Stack“ genannt.
Der Ansatz „Double Stack“ verwendet zwei Stacks. Bei dem Ansatz „Double Stack“ werden die Grundblöcke, die neu betrachtet werden müssen, auf den zweiten Stack gepusht. Sobald der erste Stack leer ist, werden die beiden Stacks vertauscht. Der Ansatz terminiert, wenn beide Stacks leer sind.
Die Ansätze „Simple Queue“ und „Priority Queue“ verwenden beide eine Queue. Sie unterscheiden sich dadurch, dass die „Priority Queue“ nach reversierter Postfix-Reihenfolge priorisiert ist, die „Simple Queue“ ist nicht priorisiert.

2.3.2. Gültige Definitionen

Wie die Analyse Lebendige Variablen zählt auch die Analyse Gültige Definitionen [9] [10] [11] mit zu den gebräuchlichsten Datenflussanalysen.
Bei dieser Analyse wird ermittelt, ob eine Definition d am Programmpunkt p gültig ist. Eine Definition d ist am Programmpunkt p gültig, wenn ein Weg vom unmittelbar drauffolgenden Programmpunkt nach p vorhanden ist, ohne dass d zerstört wird.
Eine Definition d wird zerstört, wenn eine andere Definition von d erzeugt wird. Eine Definition d wird erzeugt, wenn eine Zuweisung an d erfolgt.

Mit den aus dieser Analyse ermittelten Informationen kann beispielsweise festgestellt werden, ob es sich bei einer Variable um eine Konstante handelt.
Ein weiterer möglicher Anwendungsfall ist die Konstruktion von Use-Def- beziehungsweise Def-Use-Ketten.
Use-Def-Ketten verknüpfen die Verwendung einer Variablen mit deren Definitionen. Def-Use-Ketten verknüpfen die Definitionen einer Variablen mit deren Verwendung.
Die Datenflussgleichungen sehen folgendermaßen aus:

$$OUT(ENTRY) = BI$$

$$OUT[B] = gen \cup (IN[B] - kill)$$

$$IN[B] = \bigcup_{S\ ist\ ein\ Vorgänger\ von\ B} OUT[S]$$

Wie den Datenflussgleichungen zu entnehmen ist, handelt es sich um ein Vorwärtsproblem.
Die OUT-Menge des ENTRY-Knotens bildet die Menge BI. Im intraprozeduralen Fall ist BI die leere Menge. Im interprozeduralen Fall muss zwischen dem kontextsensitiven Fall und dem kontextinsensitiven Fall unterschieden werden.
Im kontextsensitiven Fall ist BI die Menge der Definitionen, die an der entsprechenden Aufrufstelle gültig sind. Im kontextinsensitiven Fall ist BI die Menge der gültigen Definitionen an allen zugehörigen Aufrufstellen.
Die OUT-Mengen und die IN-Mengen ergeben sich gemäß der obigen Mengengleichungen.
Die kill-Menge enthält alle Definitionen, die in B zerstört werden. Eine Definition wird zerstört, wenn eine Zuweisung an die zugehörige Variable erfolgt.
Die gen-Menge enthält alle Definitionen, die in B generiert und danach nicht zerstört werden. Eine Definition wird generiert, wenn eine Zuweisung an die zugehörige Variable erfolgt. Somit generiert jede Zuweisung eine neue Definition und zerstört alle bisherigen Definitionen der gleichen Variablen.

Unter der Annahme, dass die Definition d1 am Anfang von Basic-Block B gültig ist, ergeben sich daher folgende vier Möglichkeiten.

Fall	Lokale Informationen		Effekt auf Gültigkeit
1	$d1 \notin Gen\_B$	$d1 \notin Kill\_B$	Gültigkeit von d1 wird in B nicht beeinflusst
2	$d1 \in Gen\_B$	$d1 \notin Kill\_B$	Gültigkeit von d1 wird in B generiert
3	$d1 \notin Gen\_B$	$d1 \in Kill\_B$	Gültigkeit von d1 wird in B gekillt
4	$d1 \in Gen\_B$	$d1 \in Kill\_B$	Gültigkeit von d1 wird in B nicht beeinflusst

Tabelle 2.3.2: Fallunterscheidung Gültigkeit

Sowohl in Fall 1 als auch Fall 2 und Fall 4 ist Definition d1 am Ende von Basic-Block B gültig. Der Grund dafür ist jedoch für jeden Fall unterschiedlich. Interessant ist hier vor allem der vierte Fall, da die Gültigkeit von d1 in B gekillt, aber auch neu generiert wird. Fall 4 muss von Fall 1 und 2 unterschieden werden, da es Fälle gibt, in denen es wichtig ist zu wissen, ob die zu d1 gehörende Variable im Basic-Block B verändert wurde.
Da die Analyse Gültige Definitionen mit der Analyse Lebendige Variablen vergleichbar ist, ist damit zu rechnen, dass auch die Analyse Gültige Definitionen laufzeitmäßig davon profitiert, wenn möglichst wenig Basic-Blocks besucht werden. Es wird daher erwartet, dass auch hier die Worklist-Algorithmen am besten abschneiden werden.

2.3.3. Postdominanz

Ein Knoten a postdominiert den Knoten b, wenn jeder Pfad vom Knoten b zum EXIT-Knoten den Knoten a enthält [12].

Die Datenflussgleichungen sehen folgendermaßen aus:

$$Postdom[EXIT] = EXIT$$

$$Postdom[B] = \left(\bigcap_{S\ ist\ Nachfolger\ von\ B} Postdom[S] \right) \bigcup \{B\}$$

Wie den Datenflussgleichungen zu entnehmen ist, handelt es sich um ein Rückwärtsproblem.

Die Postdom-Menge des EXIT-Knotens enthält nur den EXIT-Knoten selbst. Die restlichen Postdom-Mengen erhält man durch den Schnitt der Postdom-Mengen der Nachfolger, vereinigt mit dem jeweiligen Knoten selbst. Schnitt deshalb, weil a auf jedem Pfad von b zum EXIT-Knoten enthalten sein muss.

Wie die Algorithmen zur Berechnung der Dominanz mit der Zeit entwickelt wurden ist in [1] beschrieben. Mit einer fast linearen Zeitschranke wird der Lengauer-Tarjan-Algorithmus als der meist bekannte und meist verwendete Algorithmus angegeben.
Die Autoren haben den Lengauer-Tarjan-Algorithmus mit dem iterativen Datenflussalgorithmus verglichen und stellten fest, dass der iterative Datenflussalgorithmus bei geschickter Wahl der Datenstrukturen, trotz schlechtem asymptotischem Verhalten, in der Praxis schneller ist.

2.3.4. Dominanz

Ein Knoten a dominiert den Knoten b, wenn jeder Pfad zum Knoten b vom ENTRY Knoten den Knoten a enthält. [9]

Die Datenflussgleichungen sehen folgendermaßen aus:

$$dom[ENTRY] = ENTRY$$

$$dom[B] = \left(\bigcap_{S\ ist\ Vorgänger\ von\ B} dom[S] \right) \bigcup \{B\}$$

Wie den Datenflussgleichungen zu entnehmen ist, handelt es sich um ein Vorwärtsproblem.

Die dom-Menge des ENTRY-Knotens enthält nur den ENTRY-Knoten selbst. Die restlichen dom-Mengen erhält man durch den Schnitt der dom-Mengen der Nachfolger, vereinigt mit jeweiligen Knoten selbst. Schnitt deshalb, weil a auf jedem Pfad zu b vom ENTRY-Knoten enthalten sein muss.

Wie [2] herausfanden profitiert die Analyse Dominanz laufzeitmäßig nicht davon, wenn möglichst wenig Basic-Blocks besucht werden.
Es wurde festgestellt, dass für kleine Graphen eine Worklist-Implementierung mit „Priority Queue“ am besten abschneidet. Für große Graphen hingegen sollen die Worklist-Implementierungen mit „Priority Queue“ und „Simple Queue“ gleicht gut sein.

Folgende Tabelle zeigt deren Messergebnisse im Überblick:

Number of Blocks	Number of Graphs	Round Robin	Single Stack	Double Stack	Priority Queue	Simple Queue
Under 50	108	1	0.7015	0.7222	0.7284	0.7068
≥ 50 & < 100	35	1	0.6277	0.6333	0.6928	0.6347
≥ 100 & < 200	13	1	0.6411	0.6449	0.7216	0.6457
≥ 200 & < 400	7	1	0.6743	0.6857	0.7657	0.6343
≥ 400	6	1	0.6936	0.7003	0.8552	0.7138
10.000	1000	1	0.3890	0.3852	0.3913	0.3834
15.000	1000	1	0.3829	0.3773	0.3778	0.3746
20.000	1000	1	0.4036	0.3881	0.3831	0.3851
25.000	1000	1	0.4387	0.4049	0.3939	0.4018

Tabelle 2.3.4 Messergebisse aus [2]

Die Zeiten sind in Sekunden angegeben.
Die Ansätze in Tabelle 2.3.4 wurden in Abschnitt 2.3.1 bereits erklärt.

2.4. Verwendete Ansätze

Die hier verwendeten Ansätze lösen die Datenflussprobleme mit Hilfe eines iterativen Fixpunkt-Algorithmus.
Dieser Algorithmus gibt nur ein Kriterium an, wann die Lösung gefunden wurde. Die Reihenfolge, in der über den Datenflussgraph iteriert wird, kann daher frei gewählt werden.
Die folgenden Ansätze unterscheiden sich deshalb lediglich in der Reihenfolge, in der die Knoten des Datenflussgraphs besucht werden.

Laut [13] funktionieren die traditionellen Algorithmen gut für einzelnen Routinen, skalieren aber nicht gut für interprozedurale Analysen. Als Grund dafür wird angegeben, dass die traditionellen Algorithmen unnötig viel Zeit damit verbringen große Teile der Programme zu reanalysieren. Die Autoren stellen einen Algorithmus vor, mit dem sie eine bis zu 90% bessere Laufzeit erzielen konnten.

2.4.1. Traditionelle iterative Depth-first-Lösung

Bei der traditionellen iterativen Depth-first-Lösung werden die Knoten des Datenflussgraphen in Depth-first-Reihenfolge durchlaufen.
Die Reihenfolge heißt Depth-first, weil versucht wird, die am weitesten vom Startknoten entfernten Knoten so schnell wie möglich zu besuchen.

Ein möglicher Algorithmus dazu sieht folgendermaßen aus:

```
function dfs(curnode)
{
   while nodes are changing do
        Visit_Praefix(curnode)
        foreach s in Successors(curnode) do
         dfs(s)
        end loop
        Visit_Postfix(curnode)
    end loop
}
```

Wie man sieht, bietet der Algorithmus grundsätzlich zwei verschiedene mögliche Besuchsreihenfolgen für die Knoten. Die Knoten können wahlweise in Präfix- oder in Postfix-Reihenfolge besucht werden. Im Rahmen dieser Diplomarbeit wird die Präfix-Variante verwendet.

Da bei der Depth-first-Traversierung die am weitesten entfernten Knoten so schnell wie möglich besucht werden, ergibt sich im Falle, dass der Graph keine Zyklen enthält, eine optimale

Besuchsreihenfolge für Vorwärtsanalysen.
Optimal deshalb, weil in diesem Fall nur eine einzige Iteration erforderlich ist. Sobald jedoch Zyklen enthalten sind, werden mehrere Iterationen notwendig, wobei bei jeder einzelnen der vollständige Graph traversiert werden muss. Dies kann sich schnell nachteilig auswirken, indem es dazu führt, dass Knoten unnötigerweise mehrfach besucht werden müssen, obwohl sich bei diesen gar nichts geändert haben kann.

2.4.2. Traditionelle iterative Lösung nach reversierter postfix-Ordnung

Bei diesem Ansatz werden die Knoten in reversierter postfix-Ordnung aufgesucht.
Ziel dieses Ansatzes ist es die Knoten in topologischer Reihenfolge zu besuchen. Ein Knoten wird daher erst dann besucht, wenn seine Vorgänger bereits besucht wurden.

Eine toplogische Reihenfolge erhält man beispielsweise indem man einmal linksherum durch den Graphen läuft, um die Knoten in Postfix-Reihenfolge zu markieren, um anschließend rechtsherum durch den Graphen zu laufen, um die vorher markierten Knoten in umgekehrter Reihenfolge zu besuchen.

Ein möglicher Algorithmus dazu sieht folgendermaßen aus:

```
function rpo(Routine)
{
nodes = getTopologicalorder(Routine)
   while nodes are changing do
      foreach n in nodes do
       Visit(n)
      end loop;
   end loop
}
```

Da bei dieser Reihenfolge jeder Knoten erst dann besucht wird, nachdem alle Vorgänger besucht wurden, ergibt sich im Falle dass der Graph keine Zyklen enthält, eine optimale Besuchsreihenfolge für Vorwärtsanalysen. Optimal deshalb, weil in diesem Fall nur eine einzige Iteration erforderlich ist.

Sobald jedoch Zyklen enthalten sind, werden mehrere Iterationen notwendig, wobei bei jeder einzelnen der vollständige Graph traversiert werden muss. Dies kann sich schnell nachteilig auswirken, indem es dazu führt, dass Knoten unnötigerweise mehrfach besucht werden müssen, obwohl sich bei diesen gar nicht geändert haben kann.

2.4.3. Worklist-Ansatz, der nur die neu zu berechnenden Grundblöcke erfasst

Die bisher betrachteten Ansätze haben alle den Nachteil, dass oft Knoten betrachtet werden, die sich nicht ändern, da bei jeder Iteration der gesamte Programmgraph iteriert wird.
Der Worklist-Ansatz betrachtet nur Knoten bei denen es Änderungen geben kann.

In [2] haben die Autoren den Worklist-Algorithmus mit unterschiedlichen Containern als Worklist evaluiert.
Die hauptsächlichen Einflussfaktoren sind zum einen die Reihenfolge, in der die Grundblöcke initial auf der Worklist landen und zum anderen, in welcher Reihenfolge die Grundblöcke die sich ändern, auf der Worklist landen. Als einfachster möglicher Container für die Worklist wird von den Autoren ein einfacher Stack genannt. Aus diesem Grund wird im Rahmen dieser Diplomarbeit ein Stack als Container für die Worklist verwendet.

Ein möglicher Algorithmus dazu sieht folgendermaßen aus:

```
function worklist(Routine)
{
    worklist = Stack(getCFGCount(Routine))
    nodes = getTopologicalorder(Routine)
    foreach n in nodes do
        worklist.Push(n)
    end loop;
    while worklist.IsNotEmpty() do
        node = worklist.Pop()
        Visit(node)
        if node has changed then
            foreach s in Successors(node) do
                if s is not in worklist then
                  worklist.Push(s)
                end if
            end loop
        end if
    end loop
}
```

Der Worklist-Ansatz hat den Vorteil, dass nur die Knoten besucht werden, die sich geändert haben können. Dies sind im Falle der Rückwärtsanalyse alle Vorgänger und im Falle der Vorwärtsanalyse alle Nachfolger des jeweils besuchten Knotens.

Da die Knoten auf der Worklist jedoch nicht sortiert sind kann es auch hier passieren, dass die Knoten nicht in optimaler Reihenfolge besucht werden. Dies kann dazu führen, dass unnötig vielen Knoten besucht werden.

2.4.4. Worklist-Ansatz, der zusätzlich die reversierte postfix-Ordnung beachtet

Wie aus dem Namen ersichtlich, wird bei diesem Ansatz zusätzlich die Worklist in reversierte postfix-Ordnung gebracht.
Ein möglicher Algorithmus dazu sieht folgendermaßen aus:

```
function worklist_rpo(Routine)
{
    worklist = Ordered_Set()
    nodes = getTopologicalorder(Routine)
    foreach n in nodes do
        worklist.Insert(n)
    end loop;
    while worklist.IsNotEmpty() do
        node = worklist.Get()
        worklist.Remove(node)
        Visit(node)
        if node has changed then
            foreach s in Successors(node) do
                if s is not in worklist then
                 worklist.Insert(s)
                end if
            end loop
        end if
    end loop
}
```

Den Nachteil des vorherigen Ansatzes versucht der zweite Ansatz zu vermeiden, indem er die Worklist in die reversierte post-fix Reihenfolge sortiert. Für den Fall, dass der Graph keine Zyklen enthält, ergibt sich so die optimale Besuchsreihenfolge, sowohl für Rückwärts- als auch Vorwärtsanalysen.

Um die Worklist in topologischer Reihenfolge zu sortieren wird die interne in den Grundblöcken gespeicherte DFS-Nummer als Sortierkriterium verwendet, da die DFS-Nummer laut [9] der topologischen Sortierung entspricht.

3. Idee

In diesem Kapitel wird die Idee zur Lösung der Aufgabenstellung behandelt.
Den Anfang macht Abschnitt 3.1 mit einer Beschreibung des Zustandes vor der Arbeit. Dies beinhaltet eine Bestandsaufnahme des IST-Zustandes, beschreibt aber auch den Entstehungsprozess der Lösungsidee.
Im Abschnitt 3.2 wird dann die Idee für die im Rahmen dieser Arbeit entstandene Implementierung vorgestellt.

3.1. Bisheriger Zustand

Die bisherige Situation sah folgendermaßen aus, dass kein einheitliches Interface für die Implementierung von Datenflussanalysen existierte.
Die vorhandenen Datenflussanalysen implementieren entweder eine eigene Strategie oder greifen auf eines der bereits vorhandenen Interfaces zurück.

Bauhaus verfügt bereits über eine Vielzahl an unterschiedlichen Interfaces die verwendet werden können, um Datenflussanalysen auf Basis von IML zu implementieren.
So gibt es zum Beispiel eine generische Implementierung des Algorithmus von Tarjan. Diese wird von einem anderen Interface verwendet, um eine Präfix- und Postfix-Traversierung des Kontrollflussgraphen zu realisieren.
Des Weiteren gibt es auch noch Interfaces für SSA-basierte Analysen.

Das Interface, welches dem für diese Arbeit benötigten am nächsten kam, war eine generische Implementierung, welche für die Lösung intraprozeduraler Datenflussprobleme verwendet werden kann. Was dieser Implementierung fehlt, ist eine Möglichkeit eine Transferfunktion anzugeben und eine Möglichkeit die Iterationsstrategie auszutauschen. Erschwerend kommt hinzu, dass diese Implementierung keine interprozeduralen Analysen ermöglicht.

Da sich die Interfaces unterscheiden, ist es nicht möglich, bei den vorhandenen Datenflussanalysen die Strategie einfach auszutauschen, da man die Datenflussanalysen jeweils für die Interfaces umschreiben müsste.

Aufgrund der fehlenden Möglichkeit die Strategien einfach auszutauschen, kommen die vorhandenen Datenflussanalysen für einen empirischen Vergleich der Strategien nicht in Frage.
Um einen empirischen Vergleich der Strategien durchführen zu können, hätte ich daher jede Datenflussanalyse mehrmals mit unterschiedlichen Ansätzen implementieren müssen.
Wie dies aussehen würde ist in Abbildung 3.1 zu sehen.

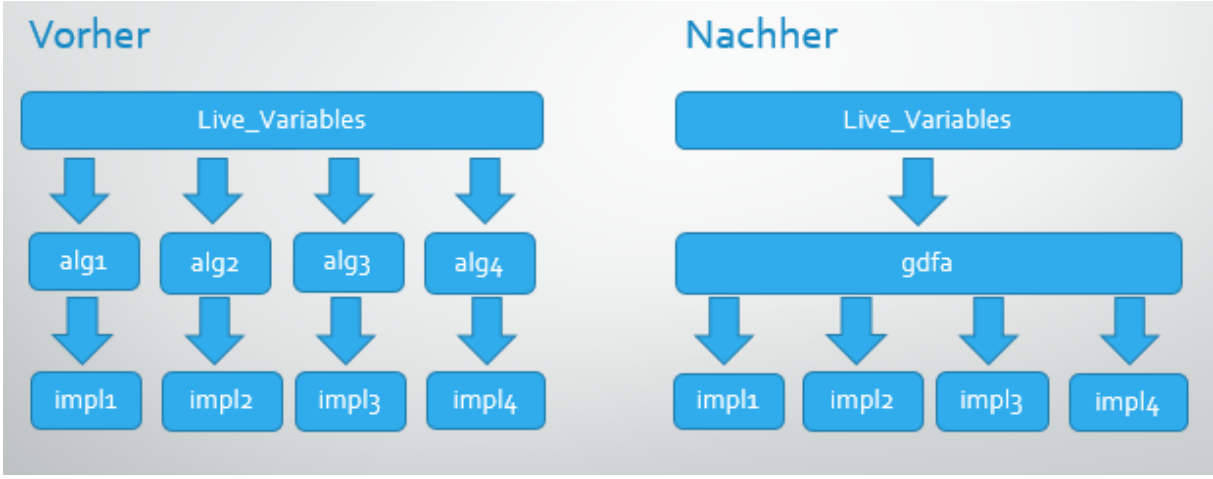

Abbildung 3.1: Zustand Vorher - Nachher

Ausgehend von diesem IST-Zustand überlegte ich mir nun eine Idee für die Umsetzung.
Das Resultat der Überlegungen ist ebenfalls in Abbildung 3.1 zu sehen.

In einem ersten Schritt setzte ich mir folgende Designziele, die ich dann bei der Implementierung verfolgt habe.
Um die unterschiedlichen Ansätze vergleichen zu können, benötigte ich eine einfache Möglichkeit, die Analysen mit unterschiedlichen Ansätzen zu implementieren.

Ich setzte mir daher zum Ziel ein generisches Skeleton zu implementieren, welches ich dann nur noch für eine konkrete Analyse instanziieren muss.
Gleichzeitig wollte ich dadurch eine einfache Möglichkeit schaffen unterschiedliche Ansätze zu vergleichen.
Meine Idee sieht nun so aus, dass ich jede Analyse nur noch einmal gegen das Skeleton zu implementieren habe und dann nur für die jeweiligen Ansätze instanziieren muss.
Dem Skeleton habe ich die Bezeichnung gdfa (genreric dataflow analysis) gegeben.

3.2. Framework in BAUHAUS

Um die unterschiedlichen Ansätze gut vergleichen zu können, wurde für diese Arbeit ein einheitliches Interface geschaffen, das nur noch für ein konkretes Datenflussproblem initialisiert werden muss. Dabei wurde der Umstand ausgenutzt, dass die meisten Analysen die gleichen Aufgaben erledigen müssen, um die Analyse von der Iterationsstrategie zu trennen. Wir unterscheiden dabei zwischen Vorwärts- und Rückwärtsanalysen.

Im Folgenden ist der Algorithmus je einmal für beide Fälle zu sehen.

Vorwärts:

$$OUT[ENTRY] = v_{ENTRY}$$

$$\textbf{for } (jeder\ Grundblock\ B\ außer\ ENTRY)\ OUT[B] = 0;$$

$$\textbf{while } (Änderungen\ an\ OUT\ erfolgen)$$

$$\textbf{for } (jeder\ Grundblock\ B\ außer\ ENTRY)\{$$

$$IN[B] = MeetFunction_{P\ ist\ ein\ Vorgänger\ von\ B} OUT[P]$$

$$OUT[B] = Transferfunction(IN[B])$$

$$\}$$

Rückwärts:

$$IN[EXIT] = v_{EXIT}$$

$$\textbf{for } (jeder\ Grundblock\ B\ außer\ EXIT)\ IN[B] = 0;$$

$$\textbf{while } (Änderungen\ an\ IN\ erfolgen)$$

$$\textbf{for } (jeder\ Grundblock\ B\ außer\ EXIT)\{$$

$$OUT[B] = MeetFunction_{P\ ist\ ein\ Nachfolger\ von\ B} IN[P]$$

$$IN[B] = Transferfunction(OUT[B])$$

$$\}$$

Dieser Algorithmus ermöglicht es, das Framework generisch zu implementieren.
Die Aufgabe des Frameworks besteht dann darin, mit der ausgewählten Strategie in der ausgewählten Richtung über die Grundblöcke zu iterieren und auf jedem Grundblock die generischen Funktionen MeetFunction und Transferfunction auszuführen.
Das Ganze wird dann für Vorwärtsanalysen solange wiederholt, bis sich kein OUT-Set mehr ändert. Für Rückwärtsanalysen wird es solange wiederholt, bis sich kein IN-Set mehr ändert.
Mit diesen Informationen konnte ich nun die Schnittstelle definieren, welche die späteren Analysen dann verwenden werden.

3.2.1. Schnittstelle

Das Skeleton benötigt folgende Werte um instanziiert zu werden:

- Direction: Vorwärts / Rückwärts
- Iterationstrategy: Depth-first / reverse Postorder / worklist / worklist rpo
- Datentyp für die Datenflusswerte
- Meet-, Process- und Transferfunktion
- Initialisierung der Datenflusswerte

Damit das Skeleton mit dem Datentyp für die Datenflusswerte umgehen kann, werden auch noch folgende Methoden benötigt.

- Destroy: gibt den Speicherplatz wieder frei
- Copy: erzeugt eine Kopie
- „=“: überprüfe ob der Wert gleich ist

Das Datentyp-Objekt, das durch die Ausführung der Meet-Funktion entsteht, wird im Folgenden als „incoming“-Set bezeichnet.
Das Datentyp-Objekt, das nach Ausführung der Process- und Meet-Funktion entsteht, wird im Folgenden als „Result“-Set bezeichnet.

Gespeichert wird für beide Richtungen jeweils das „Result“-Set, also das Ergebnis nach der Verarbeitung des entsprechenden Basic-Blocks.
Das „Result“-Set für die Routinen erhält man mit der Funktion „Get_Result_For_Routine“, das „Result“-Set für die Grundblöcke mit der Funktion „Get_Result_For_BB“.
Das „incoming“-Set für die Grundblöcke erhält man mit der Funktion „Get_In_Set_For_BB“.

3.2.2. Interprozedurale Erweiterung

Für die interprozedurale Erweiterung kommt das Konzept des interprozeduralen Kontrollflussgraphen zum Einsatz. Der interprozedurale Kontrollflussgraph verbindet den Kontrollflussgraphen der aufrufenden Routine mittels einer gerichteten Kante mit dem Kontrollflussgraphen der aufgerufenen Routine.
In der einfachsten Version wird ein Aufruf durch einen eigenen Grundblock repräsentiert. Die komplexere Version splittet diesen Grundblock in einen Call- und einen Return-Knoten auf. Wie dieses Konzept funktioniert soll an folgendem Beispiel erläutert werden.

```
int a,b,c,d;

void p()
{
     b=2;
     if(b<d)
          c=a+b;
}

void q()
{
     a=1;
     p();
     a=a*b;
}

int main()
{
     a=5;
     b=3;
     c=7;
     d=42;
     p();
     a=a+2;
     d=a*b;
     q();
     return a;
}
```

Abbildung 3.2.2.1: inter.c

Die Kontrollflussgraphen, wie sie von dem Tool „iml2cfg“ für das Programm aus Abbildung 3.2.2.1 erzeugt wurden, sind in Abbildung 3.2.2.2 visualisiert.

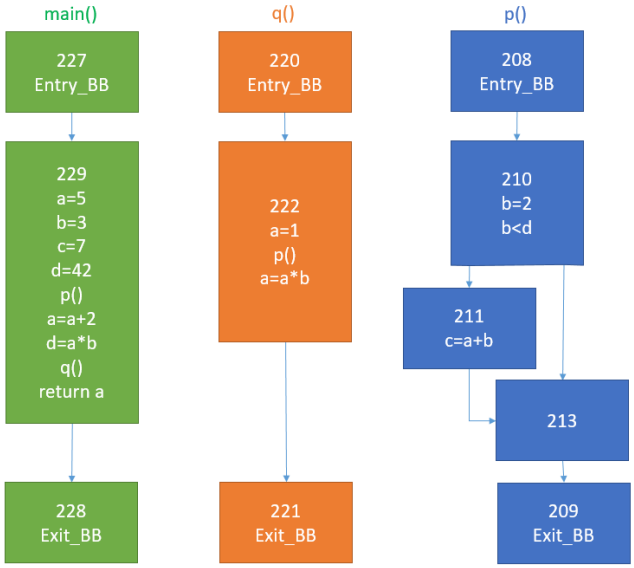

Abbildung 3.2.2.2: Kontrollflussgraphen für inter.c

Wie der zugehörige interprozedurale Kontrollflussgraph aussieht ist in Abbildung 3.2.2.3 zu sehen.

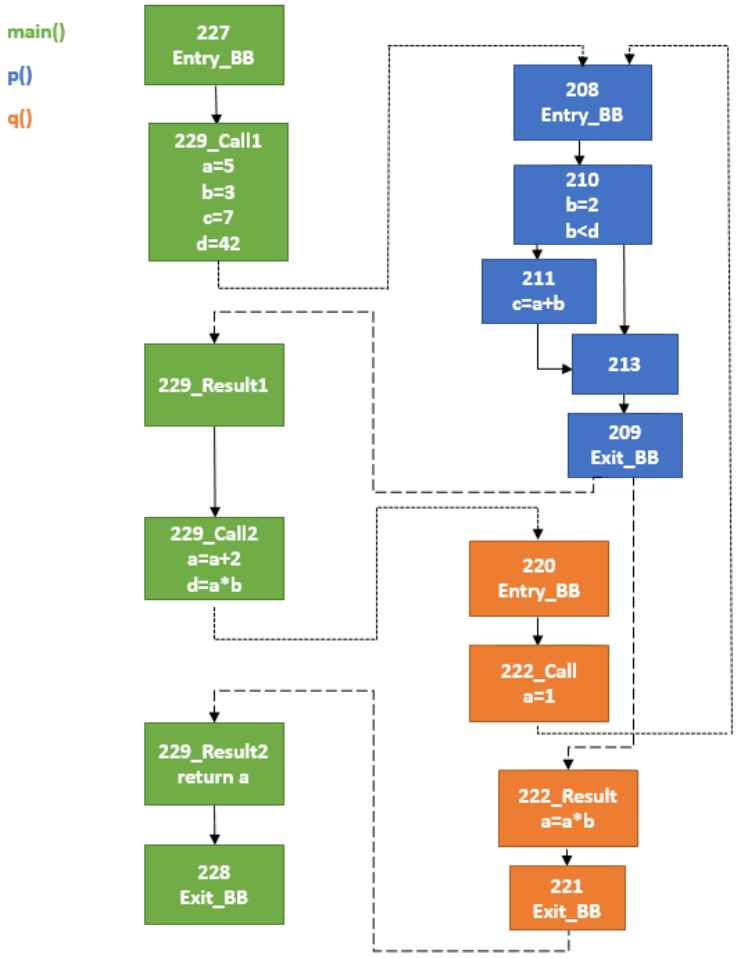

Abbildung 3.2.2.3: interprozeduraler Kontrollflussgraph für inter.c

Mit Hilfe des interprozeduralen Kontrollflussgraphen kann nun das Problem der interprozeduralen Datenflussanalyse auf das Problem der intraprozeduralen Datenflussanalyse abgebildet werden.

4. Implementierung

In diesem Kapitel werden die Implementierungen vorgestellt.
Im ersten Abschnitt wird das Skeleton, unabhängig von den später damit implementierten Analysen, beschrieben.
Im zweiten Abschnitt wird dann gezeigt, wie mit diesem Skeleton die Analysen implementiert wurden. Diese Analysen sollen auch als Beispiel dafür dienen, wie das Skeleton verwendet werden kann.

4.1. Skeleton

Das Skeleton setzt sich zusammen aus den Packages GDFA_Support, GDFA_Traversals und GDFA_Analysis. Diese Packages sollen hier nun der Reihe nach vorgestellt und erläutert werden.

4.1.1. GDFA_Support

Zuallererst werden einige Datentypen benötigt, mit denen später die Analysen instanziiert werden können.

Der Übersichtlichkeit halber entschied ich mich, diese in ein separates Package auszulagern. Dazu wird das Package „GDFA_Support“ erstellt, dessen Definition in der Abbildung 4.1.1 zu sehen ist.

```
package GDFA_Support is

type Direction_Type is (Forward, Backward);

type Iteration_Strategy_Type is
  (depth_first, reverse_post_order, worklist, rpo_worklist);

function Standard_Filter (Node : in Values.Value) return Boolean;

-- returns true if X and Y refers to the same o_node
function "="
     (X : in Entity_L_Values.Entity_L_Value;
      Y : in Entity_L_Values.Entity_L_Value)
     return Standard.Boolean;

function "<"
     (X : in Basic_Blocks.Basic_Block;
      Y : in Basic_Blocks.Basic_Block)
     return Standard.Boolean;

end GDFA_Support;
```

Abbildung 4.1.1: GDFA_Support.ads

Wie man in Abbildung 4.1.1 erkennen kann, sind in diesem Package die Types Direction_Type und Iteration_Strategy_Type definiert.

Der Direction_Type legt fest, ob es sich um eine Vorwärts- oder Rückwärtsanalyse handelt und besteht daher aus Forward und Backward. Außerdem wird der Direction_Type auch verwendet, um festzulegen, in welcher Richtung der Graph traversiert wird, dazu später mehr.

Der Iteration_Strategy_Type legt die Iterationsstrategie fest, mit welcher der Graph traversiert werden soll.
Die ebenfalls noch hier zu sehende Funktion Standard_Filter wird später bei der Traversierung der Basic_Blocks wichtig, um die Value-Knoten zu filtern.
Mit der Zeit kamen hier später noch weitere Funktionen hinzu, da es sich als praktikabler erwiesen hat diese auszulagern.
Dies sind zum einen die Funktion „=“ mit der sich für zwei Entity_L_Value-Knoten prüfen lässt, ob diese zum selben O_Variable-Knoten gehören und die Funktion „<“, die gebraucht wird um zu prüfen, ob zwei Basic_Block-Knoten topologisch vor- oder nacheinander kommen.
Mit Hilfe dieses Packages war es mir nun möglich die Traversierung zu implementieren.

4.1.2. GDFA_Traversals

Um mir die Möglichkeit offen zu lassen, später einfach weite Iterationsstrategien ergänzen zu können, habe ich Package GDFA_Traversals implementiert. Eine ursprünglich deutlich generischere Version des Packages wurde zu Gunsten der Effizienz so abgeändert, dass es nun für jede Richtung eine eigene Methode gibt. Dies führt an manchen Stellen zu Codeduplikaten, steigert jedoch die Effizienz und Lesbarkeit des Quelltextes enorm.

Die Definition dieses Packages ist in der Abbildung 4.1.2.1 zu sehen.

```
package GDFA_Traversals is

procedure Visit_All_Forward
          (Entry_BB      : Basic_Blocks.Basic_Block;
          Iteration_Strategy : GDFA_Support.Iteration_Strategy_Type;
           Do_Something : not null access procedure
          (BB : in Basic_Blocks.Basic_Block;
           BB_Changed : in out Boolean));

procedure Visit_All_Backward
          (Entry_BB      : Basic_Blocks.Basic_Block;
          Iteration_Strategy : GDFA_Support.Iteration_Strategy_Type;
           Do_Something : not null access procedure
          (BB : in Basic_Blocks.Basic_Block;
           BB_Changed : in out Boolean));

end GDFA_Traversals;
```

Abbildung 4.1.2.1: GDFA_Traversals.ads

Dieses Package stellt die Prozeduren Visit_All_Forward und Visit_All_Backward. Beide Prozeduren führen auf jedem besuchten Knoten die Prozedur Do_Something aus.

Die Prozedur Visit_All_Forward erhält den Entry-Knoten als Eingabe und traversiert den Kontrollflussgraphen mit der ausgewählten Iterationsstrategie.
Die Prozedur Visit_All_Backard erhält den Exit-Knoten als Eingabe und traversiert den inversen Kontrollflussgraphen mit der ausgewählten Iterationsstrategie.

Bei der Verwendung der Prozeduren ist darauf zu achten, dass auch wirklich der richtige Knoten übergeben wird, da sonst der Kontrollflussgraph nicht oder nur unvollständig traversiert wird.
Dieses Package kann nun vom Skeleton verwendet werden, um den Graph zu traversieren und Methoden auf den Knoten auszuführen.

Alles was dazu nötig ist, ist die entsprechende Methode GDFA_Traversals.Visit_All_Forward, bzw. GDFA_Traversals.Visit_All_Backward aufzurufen, welche dann, wie in der Abbildung 4.1.2.2 zu sehen, intern die Methode entsprechend initialisiert und aufruft.

```
procedure Visit_All_Forward
     (Entry_BB :  Basic_Blocks.Basic_Block;
      Iteration_Strategy : GDFA_Support.Iteration_Strategy_Type;
      Do_Something : not null access procedure
         (BB : in Basic_Blocks.Basic_Block;
          BB_Changed : in out Boolean))
   is
    procedure Handle_Basic_Block
         (BB : in Basic_Blocks.Basic_Block;
          BB_Changed : in out Boolean)
      is
      begin
         Do_Something (BB, BB_Changed);
      end Handle_Basic_Block;

    procedure Visit_All_Instances
      is new Generic_Visit_All_Forward
        (Iteration_Strategy, Handle_Basic_Block);
   Begin

-- We are sure we'll never get a 'No_Basic_Block'.
         pragma Assert (Entry_BB /= null);
         Visit_All_Instances(Entry_BB);
   end Visit_ALL_Forward;

procedure Visit_All_Backward
     (Entry_BB :  Basic_Blocks.Basic_Block;
         Iteration_Strategy : GDFA_Support.Iteration_Strategy_Type;
         Do_Something : not null access procedure
         (BB : in Basic_Blocks.Basic_Block;
         BB_Changed : in out Boolean))
  is
    procedure Handle_Basic_Block
              (BB : in Basic_Blocks.Basic_Block;
               BB_Changed : in out Boolean)
      is
      begin
         Do_Something (BB, BB_Changed);
      end Handle_Basic_Block;

    procedure Visit_All_Instances
      is new Generic_Visit_All_Backward
        (Iteration_Strategy,Handle_Basic_Block);
  begin

-- We are sure we'll never get a 'No_Basic_Block'.
         pragma Assert (Entry_BB /= null);
         Visit_All_Instances(Entry_BB);
  end Visit_ALL_Backward;
```

Abbildung 4.1.2.2: Initialisierung Visit_All_Instance

Die eigentlichen Traversierungen sind in den Methoden Generic_Visit_All_Forward, bzw. Generic_Visit_All_Backward implementiert. Um eine weitere Iterationsstrategie mit aufzunehmen, müssen nur diese Methoden entsprechend erweitert werden.
Die Methoden iterieren solange immer wieder neu über den Graphen, solange BB_Changed bei mindestens einem Basic_Block true zurück geliefert hat.

4.1.3. GDFA_Analysis

In diesem Abschnitt wird der Hauptteil der Implementierung behandelt, das eigentliche Skeleton.

Das Skeleton wird in Form des generischen Package GDFA_Analysis geliefert.
Das Package GDFA_Analysis verwendet das Package GDFA_Traversals, um über die Kontrollflussgraphen zu iterieren und diese dabei zu analysieren.

Damit das Skeleton weiß was es tun soll, muss es zuerst initialisiert werden.
Sehen wir uns daher als erstes die Schnittstelle an, die implementiert werden muss, um eine Analyse zu programmieren. Diese Schnittstelle ist in der Abbildung 4.1.3.1 zu sehen.

Zuerst werden hier die Richtung der Analyse und die zu verwendende Iterationsstrategie definiert. Als nächstes wird ein Type Collected_Data definiert. Der Type Collected_Data dient als Type für die von der Analyse zu sammelnden Fakten. Auf diesem Type arbeiten dann auch die meisten der nachfolgenden Funktionen und Prozeduren.

Mit der Funktion Filter wird bestimmt, ob ein Value-Knoten beachtet werden soll oder nicht.
Als nächstes folgt die Prozedur Transfer_Function, welche verwendet wird, um in Abhängigkeit von den in einem Collected_Data-Objekt gespeicherten Informationen, Änderungen an diesem Collected_Data-Objekt vorzunehmen.

Die Prozedur Process wird verwendet, um in Abhängigkeit der Value-Knoten Änderungen an den Collected_Data-Objekten vorzunehmen.
Die Prozedur Meet_Function implementiert je nach Analyse entweder eine Union- oder eine Intersection-Operation auf dem Type für das Collected_Data-Objekt.
Die Prozedur Pre_Prozess wird verwendet, um notwendige Änderungen an einem Collected_Data-Objekt vorzunehmen, die vor der Anwendung der Prozedur Prozess anfallen.

Die Funktion Start_Set initialisiert den Type für das Collected_Data-Objekt für jeden BasicBlock. Je nach Analyse wird das Collected_Data-Objekt entweder als vollständiges oder als leeres Set initialisiert.
Die Funktion Destroy_Set dient zum Aufräumen und wird eingesetzt, um den Speicher für die Fakten wieder freigeben zu können.
Die Funktion Copy_Set dient zum Kopieren und wird eingesetzt, um eine Kopie der Fakten erzeugen zu können.

Hinzu kommen noch zwei Methoden, die für die interprozedurale Datenflussanalyse notwendig sind. Die Prozedur Remove_Locals filtert die lokalen Variablen aus den Ergebnissen für die einzelnen Routinen heraus.
Die Funktion Connector_Finished wird verwendet, um zu prüfen, ob eine Routine noch ein weiteres Mal analysiert werden muss.

```
generic
   Direction : GDFA_Support.Direction_Type;
     Iteration_Strategy : GDFA_Support.Iteration_Strategy_Type;

-- type for the facts to collect with that analysis
   type Collected_Data is private;
   with function "="
     (A : in Collected_Data;
      B : in Collected_Data)
     return Boolean is <>;

-- used to filter local variables in case of
-- interprecedural analysis
   with procedure Remove_Locals(Data : in out Collected_Data;
                                Routine : in Routines.Routine);

   with function Connector_Finished
     (A : in Collected_Data;
      B : in Collected_Data)
     return Boolean is <>;

-- shall return True if Process (Node) should be called
   with function Filter (Node : in Values.Value)
     return Boolean
   is GDFA_Support.Standard_Filter;

-- calculates the outcome of the transferfunction
   with procedure Transfer_Function (Data: in out Collected_Data);

   -- process a node (kill and gen operation)
   with procedure Process
     (Node : in     Values.Value;
      Data : in out Collected_Data);

-- depending on Kind, this should be Intersect or Union
-- used to combine Collected_Datas of preceding basic blocks
   with procedure Pre_Process
     (A : in out Collected_Data;
      B : in     Collected_Data);

-- depending on Kind, this should be Intersect or Union
-- used to combine Collected_Datas of preceding basic blocks
   with procedure Meet_Function
     (A : in out Collected_Data;
      B : in     Collected_Data);

-- depending on Kind, this should be Empty_Set or Full_Set
   with function Start_Set (BB : in Basic_Blocks.Basic_Block)
     return Collected_Data;

-- we need routine to copy and destroy Collected_Datas
   with procedure Destroy_Set (Data : in out Collected_Data);
   with function Copy_Set (Data : in Collected_Data)
     return Collected_Data;

-- debug output for in- and out-sets
   with procedure Dump_Set (Data : in Collected_Data) is null;

package GDFA_Analysis is
```

Abbildung 4.1.3.1: Interface

Soviel zur Schnittstelle, kommen wir nun zum eigentlich Package und wie es funktioniert. Dieses ist in nachfolgender Abbildung 4.1.3.2 zu sehen.

```
package GDFA_Analysis is

-- execute the dataflow analysis
-- visit the callgraphroot first and follow the calls
-- basic_blocks must have been created before calling this function!
-- don't forget to call Destroy afterwards, when the results are han-
dled
   procedure Perform_Analysis (Graph : in IML_Graphs.IML_Graph);

-- returns the "Result" set for the exit block
--(if Direction = Forward)
-- or the "Result" set for the entry block (if Direction = Backward)
-- caller must not destroy the return value
   function Get_Result_For_Routine (Routine : in Routines.Routine)
                                    return Collected_Data;

-- returns the "Result" set
-- caller must not destroy the return value
   function Get_Result_For_BB (BB : in Basic_Blocks.Basic_Block)
                               return Collected_Data;

-- returns the "incoming" set
-- caller *must* destroy the return value
   function Get_In_Set_For_BB (BB : in Basic_Blocks.Basic_Block)
                               return Collected_Data;

-- you must call this to free allocated ressources;
-- note that this also destroys the Collected_Datas, i.e. it is a
-- deep destroy
   procedure Destroy;

end GDFA_Analysis;
```

Abbildung 4.1.3.2: GDFA_Analysis.ads

Die Hauptmethode des Package ist die Prozedur Perform_Analysis. Diese Prozedur erhält den IML-Graphen als Eingabe, iteriert über alle Routinen und traversiert für jede Routine den zugehörigen Kontrollflussgraphen mit Hilfe des Packages GDFA_Traversals.

Während der Traversierung wird auf jedem Knoten die Methode Visit_BB aufgerufen, innerhalb der die eigentlich Analyse mit Hilfe der eben besprochenen Methoden stattfindet.

Die Methode Visit_BB ist in der Abbildung 4.1.3.3 zu sehen.

```
procedure Visit_BB
      (BB          : in     Basic_Blocks.Basic_Block;
       BB_Changed : in out Boolean)
   is
      Node : Values.Value;
      Facts : Collected_Data;
      Call : Direct_Calls.Direct_Call;
      O : O_Routines.O_Routine;
      R : Routines.Routine;
   begin
     BB_Changed := False;

-- process incoming sets
      Facts := Get_In_Set_For_BB (BB);

-- process statements in BB
      Node := Get_First_Value (BB);

      while Node /= null loop
         if Filter (Node) then
            Process (Node, Facts);
         end if;

      if Node.all in Direct_Calls.Direct_Call_Class'Class then
        Call := Direct_Calls.Direct_Call(Node);
        O := Direct_Calls.Get_Routine_Declaration(Call);
        R := O_Routines.Get_HPG_Routine(O);
        Process_Callee(R,Facts);
      end if;
         Node := Get_Next_Value (Node);
      end loop;

      Transfer_Function(Facts);

      if Facts /= Get_BB_Set (BB) then
         BB_Changed := True;
         Set_BB_Set (BB, Facts);
      else
         null;
      end if;
   end Visit_BB;
```

Abbildung 4.1.3.3: Implementierung Visit_BB

Zunächst wird mit der Funktion Get_In_Set_For_BB, in Abhängigkeit der Richtung der Analyse, das incoming Collected_Data-Objekt bestimmt.
Die Funktion Get_In_Set_For_BB verwendet intern die Prozeduren Meet_Function und Pre_Process. Anschließend wird über die Value-Knoten des BB iteriert und dabei für alle Value-Knoten, für welche die Funktion Filter true zurück liefert, die Methode Process aufgerufen. Auf dem daraus resultierenden Collected_Data-Objekt wird nun die Methode Transfer_Function aufgerufen.

Abschließend wird das neu berechnete „Result“-Set mit dem zuvor berechneten „Result“-Set verglichen und falls Änderungen vorhanden sind, wird das alte „Result“-Set mit dem neuen „Result“-Set überschrieben.
Die nachfolgenden Funktionen funktionieren in Abhängigkeit zu der Richtung der Analyse. So liefert die Funktion Get_Result_For_Routine für Vorwärtsanalysen das „Result“-Set für

den Exit-Block zurück. Für Rückwärtsanalysen liefert die Funktion Get_Result_For_Routine das „Result“-Set für den Entry-Block zurück.

Die Funktion Get_Result_For_BB liefert für den entsprechenden Basic-Block das zugehörige „Result“-Set zurück. Die Funktion Get_In_Set_For_BB liefert für den entsprechenden Basic-Block das zugehörige „Incoming“-Set zurück.
Im Gegensatz zum „Result“-Set wird das „Incoming“-Set bei jedem Aufruf, unter Anwendung der vorhin erwähnten Methode Meet_Function, aus den Fakten aller Vorgänger bzw. Nachfolger neu gebildet. Die Prozedur Destroy schließlich wird verwendet, um nach der Analyse wieder aufzuräumen.

Damit die Analysen auch interprozedural laufen können, wurde ein zusätzlicher Filter für Direct_Call-Knoten eingebaut, der für jeden Call die Methode Process_Callee ausführt. Die Methode Process_Callee ist in Abbildung 4.1.3.4 zur sehen.

```
procedure Process_Callee(Routine : in Routines.Routine; Data :
   in out Collected_Data) is
 Copy : Collected_Data;
 EB : Basic_Blocks.Basic_Block;

     procedure Visit_CB
         (BB         : in     Basic_Blocks.Basic_Block;
         BB_Changed :    in out Boolean)
         Is
          ...
    end Visit_CB;
begin

     if  Routines."/="(Routine,null)  then
         Transfer_Function(Data);
         EB := Get_Entry(Routine);
         Copy := Get_BB_Set(EB);
         Meet_Function(Copy,Data);
         if not Connector_Finished(Get_BB_Set(EB),Copy)
         or not  Routines.Sets.Is_Member(Seen2,Routine)  then
             Routines.Sets.Insert(Seen2,Routine);
             Set_BB_Set(EB,Copy);
                 if not  Routines.Sets.Is_Member(Seen,Routine)
                 then
                 Routines.Sets.Insert(Seen,Routine);

                 end if;
                 if Direction = Forward then
                  GDFA_Traversals.Visit_All_Forward(EB,
                        Iteration_Strategy,
                        Visit_CB'Access);
                    else
                    GDFA_Traversals.Visit_All_Backward(EB,
                     Iteration_Strategy,
                      Visit_CB'Access);
                 end if;
         end if;
         Data := Get_BB_Set(Get_Exit(Routine));
         Remove_Locals(Data,Routine);
         Copy := Start_Set(EB);
         Meet_Function(Copy,Data);
         Data := Copy;
     end if;
     if Routines.Sets.Is_Member(Seen,Routine)  then
         Routines.Sets.Remove(Seen,Routine);
     end if;
end Process_Callee;
```

Abbildung 4.1.3.4: Implementierung Process_Callee

Die Methode Process_Callee simuliert das Vorhandensein eines interprozeduralen Kontrollflussgraphen, teilt den Basic-Block BB also logisch auf in BB-Call und BB-Result. BB-Call enthält alle Statements vor dem Call. BB-Result enthält alle Statements nach dem Call.

Die Methode Process_Callee berechnet daher ein „Result“-Set für den BB-Call und propagiert dieses weiter an die aufgerufene Routine. Anschließend wird die Routine analysiert. Das Ergebnis der Analyse wird dann an den BB-Result propagiert. Dannach arbeitet die Visit_BB Methode auf dem BB-Result weiter.

Die Methode Process_Callee muss jede Routine für jeden Aufrufkontext einmal analysieren, da erst dann das „Incoming“-Set für die Routine vollständig ist. Es wird daher für jede Routine mit der Methode Connector_Finished geprüft, ob zu dem „Incoming“-Set noch etwas hinzugekommen ist. Wenn dies nicht der Fall ist, wird die Routine nicht noch einmal reanalysiert.

Um mit Zyklen umgehen zu können, wird mit Seen eine Liste der Routinen geführt, die gerade besucht werden. Mit Seen2 wird noch eine weitere Liste geführt, die dafür sorgen soll, dass jede Routine die aufgerufen wird, mindestens einmal analysiert wird.

4.2. Analysen

Das Skeleton bietet zwei Möglichkeiten, um Analysen zu implementieren.
Die erste Möglichkeit besteht darin, direkt mit der Process Methode, in Abhängigkeit der Value-Knoten, das „Result“-Set für den entsprechenden Basic_Block zu berechnen. In diesem Fall würde die Methode Transfer_Function dann einfach nichts tun.

Bei der zweite Methode werden zuerst mit der Process Methode Informationen gesammelt und diese dann im Collected_Data-Objekt gespeichert. Anschließend wird auf Basis dieser Informationen, mit Hilfe der Methode Transfer_Function, das „Result“-Set berechnet.
Für die Implementierung der nachfolgenden Analysen wurde die zweite Methode verwendet.

4.2.1. Live_Variables

Der Deklarationsteil ist in folgender Abbildung 4.2.1. zu sehen.

```
package Live_Variables_Analysis is

package Sets is new Ordered_Sets (Item_Type =>
Entity_L_Values.Entity_L_Value,
"=" =>  Entity_L_Values."=",
"<" =>  Entity_L_Values.Less);

type live_variables_type is
record
UseSET : Sets.Set;
DefSET : Sets.Set;
INSET : Sets.Set;
OUTSET : Sets.Set;
end record;

-- true for A.INSET = B.INSET
function "="
(A : in live_variables_type;
 B : in live_variables_type)
return Boolean;

procedure Remove_Locals(Data : in out live_variables_type;
                        Routine : in Routines.Routine);
```

```
-- true for A.OUTSET = B.OUTSET
function Connector_Finished
(A : in live_variables_type;
 B : in live_variables_type)
return Boolean;

-- true for Node /= null and then Node.all in
-- Entity_L_Values.Entity_L_Value_Class'Class;
function Filter (Node : in Values.Value)
return Boolean;

-- Calculates Data.INSET =
-- Union(Data.UseSET,Diff(Data.OUTSET,Data.DefSET))
procedure Transfer_Function (Data: in out live_variables_type);

-- Calculates Data.UseSET and Data.DefSET
procedure Process
(Node : in Values.Value;
 Data : in out live_variables_type);

-- Calculates Union(A.OUTSET,B.INSET)
procedure Pre_Process
(A : in out live_variables_type;
 B : in live_variables_type);
-- Calculates Union(A.OUTSET,B.INSET)
procedure Merge
(A : in out live_variables_type;
 B : in live_variables_type);

-- init all sets as empty set
function Start_Set (BB : in Basic_Blocks.Basic_Block)
return live_variables_type;
procedure Destroy_Set (Data : in out live_variables_type);
function Copy_Set (Data : in live_variables_type)
return live_variables_type;
procedure Dump_Set (Data : in live_variables_type);

end Live_Variables_Analysis;
```

Abbildung 4.2.1: Live_Variables_Analysis.ads

Das Collected_Data-Objekt wird hier als Record realisiert, der die Sets UseSET, DefSET, INSET und OUTSET enthält. Initialisiert wird das Collected_Data-Objekt mit 4 leeren Sets.
Bei Berechnung des „Incoming“-Set wird mit der Merge-Methode das Set OUTSET befüllt.
Die Process-Methode berechnet dann die Sets UseSET und DefSET.

Bei der Bestimmung der Sets UseSET und DefSET wird folgendermaßen vorgegangen.
Zunächst wird mit der Funktion Filter sichergestellt, dass die Process-Methode nur auf Entity_L_Value-Knoten ausgeführt wird.
Die Process-Methode prüft dann, ob der Entity_L_Value-Knoten einen Read- oder Assignment-Knoten als Parent hat.

Wenn es sich bei dem Parent um einen Read-Knoten handelt, dann wird der Entity_L_Value-Knoten dem UseSET hinzugefügt. Falls es sich bei dem Parent um einen Assignment-Knoten handelt, dann wird die Sache komplizierter. Zum einen muss der Entity_L_Value-Knoten dem DefSET hinzugefügt werden und zum anderen müssen alle Entity_L_Value-Knoten aus dem UseSET entfernt werden, die auf denselben O_Variable-Knoten verweisen wie der Entity_L_Value-Knoten, der gerade zum DefSET hinzugefügt wurde.

Die Pre_Process-Methode sorgt dafür, dass im „Incoming“-Set alle Sets außer dem OUTSET leer sind. Anschließend berechnet dann die Transfer_Function Methode das Set INSET.

4.2.2. Reaching_Defs

Der Deklarationsteil ist in folgender Abbildung 4.2.2 zu sehen.

```
package Reaching_Defs_Analysis is

package Sets is new Ordered_Sets (Item_Type =>
Entity_L_Values.Entity_L_Value,
"=" =>  Entity_L_Values."=",
"<" =>  Entity_L_Values.Less);
package OSets is new Ordered_Sets (Item_Type =>
O_Variables.O_Variable,"=" =>  O_Variables."=",
"<" =>  O_Variables.Less);

type reaching_defs_type is
record
Gen : Sets.Set;
Kill : OSets.Set;
INSET : Sets.Set;
OUTSET : Sets.Set;
end record;

-- true for A.OUTSET = B.OUTSET
function "="
(A : in reaching_defs_type;
 B : in reaching_defs_type)
return Boolean;

procedure Remove_Locals(Data : in out reaching_defs_type;
                        Routine : in Routines.Routine);

-- true for A.INSET = B.INSET
function Connector_Finished
(A : in reaching_defs_type;
 B : in reaching_defs_type)
return Boolean;

-- true for Node /= null and then Node.all in
-- Entity_L_Values.Entity_L_Value_Class'Class;
function Filter (Node : in Values.Value)
return Boolean;

-- Calculates Data.OUTSET =
-- Union(Data.Gen,Diff(Data.INSET,Data.Kill))
procedure Transfer_Function (Data: in out reaching_defs_type);

-- Calculates Data.Gen and Data.Kill
procedure Process
(Node : in Values.Value;
 Data : in out reaching_defs_type);

-- Calculates Union(A.INSET,B.OUTSET)
procedure Initial_Merge
(A : in out reaching_defs_type;
 B : in reaching_defs_type);

-- Calculates Union(A.INSET,B.OUTSET)
procedure Merge
(A : in out reaching_defs_type;
 B : in reaching_defs_type);
```

```
-- init all sets as empty set
function Start_Set (BB : in Basic_Blocks.Basic_Block)

return reaching_defs_type;
procedure Destroy_Set (Data : in out reaching_defs_type);
function Copy_Set (Data : in reaching_defs_type)
return reaching_defs_type;
procedure Dump_Set (Data : in reaching_defs_type);
end Reaching_Defs_Analysis;
```

Abbildung 4.2.2: Reaching_Defs_Analysis.ads

Das Collected_Data-Objekt wird hier als Record realisiert, der die Sets Gen, Kill, INSET und OUTSET enthält. Initialisiert wird das Collected_Data-Objekt mit 4 leeren Sets.
Bei Berechnung des „Incoming“-Set wird mit der Merge-Methode das Set INSET befüllt.

Die Methode Process berechnet die Sets Gen und Kill. Bei der Bestimmung der Sets Gen und Kill wird folgendermaßen vorgegangen. Zunächst wird mit der Funktion Filter sichergestellt, dass die Process-Methode nur auf Entity_L_Value-Knoten ausgeführt wird.
Die Process-Methode prüft dann, ob der Entity_L_Value-Knoten einen Assignment-Knoten als Parent hat.
Wenn der Entity_L_Value-Knoten einen Assignment-Knoten als Parent hat, dann wird der. Entity_L_Value-Knoten dem Set Gen hinzugefügt.

Zusätzlich wird dem Set Kill der O_Variable-Knoten, auf den der zum Set Gen hinzugefügte Entity_L_Value-Knoten verweist, hinzugefügt.
Die Pre_Process-Methode sorgt dafür, dass im „Incoming“-Set alle Sets außer dem INSET leer sind.

Anschließend berechnet dann die Transfer_Function-Methode das Set OUTSET.
Die Methode Transfer_Function geht dabei folgendermaßen vor. Sie erzeugt eine Kopie des Sets INSET. Dann prüft die Methode für alle in dieser Kopie enthaltenen Entity_L_Value-Knoten, ob diese auf einen der im Set Kill enthaltenen O_Variable-Knoten verweisen.
Wenn dies der Fall ist, wird der entsprechende Entity_L_Value-Knoten aus der Kopie des Sets INSET entfernt.

4.2.3. Post_Dom

Der Deklarationsteil ist in folgender Abbildung 4.2.3 zu sehen.

```
package Post_Dom_Analysis is

package Sets is new Ordered_Sets (Item_Type =>
Basic_Blocks.Basic_Block,"=" =>  Basic_Blocks."=",
"<" =>  Basic_Blocks.Less);

type post_dom_type is
record
BB : Basic_Blocks.Basic_Block;
Postdom: Sets.Set;
end record;

function "="
(A : in post_dom_type;
 B : in post_dom_type)
return Boolean;

-- true for  Node /= null
function Filter (Node : in Values.Value)
return Boolean;

-- Calculates Sets.Insert(Data.Postdom,Data.BB)
procedure Transfer_Function (Data: in out post_dom_type);

-- Calulates Data.Postdom and Data.BB
procedure Process
(Node : in Values.Value;
 Data : in out post_dom_type);

--Calculates Sets.Intersection(A.Postdom,B.Postdom);
procedure Initial_Merge
(A : in out post_dom_type;
 B : in post_dom_type);

--Calculates Sets.Intersection(A.Postdom,B.Postdom);
procedure Merge
(A : in out post_dom_type;
 B : in post_dom_type);

-- init all sets as full set
function Start_Set (BB : in Basic_Blocks.Basic_Block)
return post_dom_type;
procedure Destroy_Set (Data : in out post_dom_type);
function Copy_Set (Data : in post_dom_type)
return post_dom_type;
procedure Dump_Set (Data : in post_dom_type);
end Post_Dom_Analysis;
```

Abbildung 4.2.3: Post_Dom_Analysis.ads

Das Collected_Data-Objekt wird hier als Record realisiert, der das Set Postdom und den Basic_Block BB enthält. Initialisiert wird das Collected_Data-Objekt mit einem Fullset und BB, als zugehörigen Basic_Block. Bei Berechnung des „Incoming“-Set wird mit der Merge-Methode das Set Postdom entsprechend gekürzt. Die Process-Methode prüft dann für jeden Value-Knoten, ob der zugehörige Basic_Block bereits enthalten ist und fügt ihn hinzu, falls er fehlt. Die Transfer_Function-Methode fügt den Basic_Block selbst zum Set Postdom hinzu.

4.2.4. Dom

Der Deklarationsteil ist in folgender Abbildung 4.2.4 zu sehen.

```
package Dom_Analysis is

package Sets is new Ordered_Sets (Item_Type =>
Basic_Blocks.Basic_Block,"=" =>  Basic_Blocks."=","<" =>
Basic_Blocks.Less);

type dom_type is
record
BB : Basic_Blocks.Basic_Block;
Dom: Sets.Set;
end record;

 function "="
(A : in dom_type;
 B : in dom_type)
return Boolean;

--- true for  Node /= null
function Filter (Node : in Values.Value) return Boolean;
--- Calculates Sets.Insert(Data.dom,Data.BB)
procedure Transfer_Function (Data: in out dom_type);
--- Calulates Data.Postdom and Data.BB
procedure Process
(Node : in Values.Value;
Data : in out dom_type);
----Calculates Sets.Intersection(A.dom,B.dom);
procedure Initial_Merge
(A : in out dom_type;
 B: in dom_type);
----Calculates Sets.Intersection(A.dom,B.dom);
procedure Merge
(A : in out dom_type;
 B: in dom_type);

--- init all sets as full set
function Start_Set (BB : in Basic_Blocks.Basic_Block) return dom_type;
procedure Destroy_Set (Data : in out dom_type);
function Copy_Set (Data : in dom_type) return dom_type;
procedure Dump_Set (Data : in dom_type);
end Dom_Analysis;
```

Abbildung 4.2.4: Dom_Analysis.ads

Das Collected_Data-Objekt wird hier als Record realisiert, der das Set Dom und den Basic_Block BB enthält. Initialisiert wird das Collected_Data-Objekt mit einem Fullset und BB, als zugehörigen Basic_Block.

Bei Berechnung des „Incoming“-Set wird mit der Merge-Methode das Set Dom entsprechend gekürzt. Die Process-Methode prüft dann für jeden Value-Knoten, ob der zugehörige Basic_Block bereits enthalten ist und fügt ihn hinzu, falls er fehlt.

Die Transfer_Function-Methode schließlich fügt den Basic_Block selbst zum Set Dom hinzu.

5. Empirische Evaluation

In diesem Kapitel wird der Kontext der empirischen Evaluation der unterschiedlichen Ansätze beschrieben. Zu Beginn diskutieren wir, welche Aspekte wir evaluieren wollen.
Anschließend wird die Methodik beschrieben, mit welcher wir diese Ziele erreichen wollen. Dies beinhaltet eine Beschreibung der zu analysierenden Programme und eine Beschreibung des Testsystems.
Die nachfolgenden Abschnitte stellen dann die Resultate der Versuche vor und besprechen diese nachfolgend.

5.1. Testaufbau

Zu allererst möchte ich auf die Ziele der Evaluation eingehen.
Um zu möglichst guten Ergebnissen zu kommen ist es wichtig, nicht einfach nur das zu messen, was einfach zu messen ist. Wir müssen uns deshalb daher vorab überlegen, welche Aspekte evaluiert werden sollten.

Diese sind:

- **Korrektheit der Implementierung*:*** *Es ist wichtig zu zeigen, dass die Implementierung die korrekten Ergebnisse liefert. Erst dann ergibt es Sinn, das eigentliche Ziel in Angriff zu nehmen, nämlich die Untersuchung der Auswirkungen der unterschiedlichen Ansätze*.
- **Auswirkung der Ansätze**: *Die Evaluation soll untersuchen, welche Auswirkungen die unterschiedlichen Ansätze auf die Ausführungscharakteristika der unterschiedlichen Datenflussanalysen hat.*

5.1.1. Korrektheit

Die Korrektheit der Implementierung wurde mit folgenden Maßnahmen überprüft:

- handgeschriebene, kleine Testprogramme, um die grundsätzliche Korrektheit sicher zu stellen
- reale Programme, um zu prüfen, ob die Analysen für jeden Ansatz dieselben Ergebnisse liefern
- reale Programme, um abzuschätzen, ob die Größe der Ergebnismengen in etwa der moderner Analysen entspricht

Es ist klar, dass diese Maßnahmen keine absolute Sicherheit geben können, sondern lediglich Fehler aufdecken können.
Ein vollständiger Beweis der Korrektheit für die Implementierung liegt jenseits des Machbaren. Das hier gewählte Vorgehen entspricht jedoch bekannten, anerkannten Strategien.

5.1.2. Metriken und Methodik

Um die Ausführungscharakteristika zu vermessen, mussten geeignete Metriken gewählt werden. Anschließend wurde eine geeignete Methodik gewählt, um diese Metriken zu erheben.

Sowohl die Metriken als auch die zugehörige Methodik werden im Folgenden beschrieben. Von vorneherein fest standen die Metriken Laufzeit und Speicherverbrauch.

Für die Erfassung von Laufzeit und Speicherverbrauch kommen die Implementierungen von Staiger-Stöhr [12] zum Einsatz. Die zugehörige Schnittstelle ist in Abbildung 5.1.2 zu sehen.

```
--
--  The Bauhaus Project
--    http://www.bauhaus-stuttgart.de/
--    University of Stuttgart, Germany.
--    University of Bremen, Germany.
--    TTI GmbH TGZ Softwareanalysen.
--    http://www.axivion.com/
--    Axivion GmbH.
--
--  Copyright (c) 2009 University of Stuttgart,
--                     Department of Programming Languages
--
--  First author: Stefan Staiger-Stoehr
--
--  $Author$
--  $Date$
--  $Revision$
--
--  Purpose:
--    Measure some statistics.

with Globals;

package Statistics is

   procedure Start_Time_Measuring;

   function Stop_Time_Measuring
     return Globals.UInt;
   -- return value = ms since last Start_Time_Measuring

   function Start_Duration
      return Natural;

   function Stop_Duration
     (Key : in Natural)
     return Globals.UInt;

   procedure Stop_And_Show_Time;
   -- prints "Time : X ms"

   procedure Show_Time
     (Duration : in Globals.UInt);

   function Get_Memory
      return Globals.UInt;

   function Get_Memory_In_MB
     (Amount : in Globals.UInt := 0)
      return String;

--    procedure Write_CSV_Header;
--
--    procedure Write_Stats_To_CSV;

end Statistics;
```

Abbildung 5.1.2: Statistics.ads

Die Implementierung zur Erfassung der Laufzeit verwendet die vom Linux-Betriebssystem zur Verfügung gestellte „Uhr“ in Jiffies und rechnet diese Einheit in Millisekunden um.

Um Schwankungen bei der Messung der Laufzeit so weit wie möglich zu vermeiden, wurde der Testrechner fast ausschließlich für die Durchführung der Analysen verwendet. Abweichungen durch unterschiedliche Belastung des Rechners wird damit so gut wie ausgeschlossen.

Die Messgenauigkeit für die Laufzeit beträgt ein Jiffy. Ein Jiffy entspricht umgerechnet 10 Millisekunden. Die Ergebnisse für die Laufzeit sind deshalb auf 10 ms genau.

Die Implementierung zur Erfassung des Speicherverbrauchs verwendet zur Ermittlung des Speicherverbrauchs die Prozess-Informationen des Betriebssystems. Diese Implementierung berechnet jedoch nicht von sich aus die richtigen Werte. Um zum tatsächlichen Speicherverbrauch zu kommen, muss der Speicherverbrauch an allen relevanten Stellen erfasst werden.

Dadurch kann am Ende das Ergebnis so bereinigt werden, dass nur jener Speicherverbrauch erfasst wird, der tatsächlich durch die Analyse neu entsteht.
Die Messgenauigkeit beim Speicherverbrauch beträgt ein MB. Die bedeutet, dass Werte die mit null MB gemessen wurden in Wahrheit zwischen null und zwei MB liegen können.

Da diese Metriken jedoch beide sehr von der für die Messung verwendeten Hardware abhängen und unklar ist, ob die Unterschiede in der Laufzeit nicht zu gering ausfallen würden, habe ich mich dazu entschieden, noch zwei hardwareunabhängige Metriken hinzuzunehmen.

Die erste solche Metrik, die ich hinzugenommen habe, ist die Besuchsrate der Grundblöcke. Von dieser Metrik verspreche ich mir insbesondere in Kombination mit Laufzeit und Speicherverbrauch die Möglichkeit, einige interessante Aussagen über die Auswirkungen der Ansätze.
Die zweite Metrik ist die Ausführungshäufigkeit der Transferfunktionen.

5.1.3. Testsystem

Die Experimente fanden in einer Parallels Desktop 11 for Mac VM statt, der 2 Intel Core i7 Prozessoren, die jeweils mit 2.8 GHz getaktet sind, und 8 GB Arbeitsspeicher zugewiesen wurden.

Das Betriebssystem der VM ist Debian Linux. Da es sich bei dem Betriebssystem um eine 32-Bit-Version handelt, stehen dem System effektiv nur 2,7 GB Arbeitsspeicher zur Verfügung.

Da keine Parallelisierung vorgenommen wurde, laufen die Analysen nur auf einem Prozessor. Von der Möglichkeit, der VM noch mehr Prozessoren zuweisen, wurde daher abgesehen, da davon unter diesen Umständen keine Verbesserung zu erwarten war.

5.1.4. Testprogramme

Nachfolgend findet sich eine kurze Beschreibung der Testprogramme. Es wurde für jedes Programm eine Auswahl an Charakteristiken erfasst. Die Auswahl der Charakteristiken erfolgte in Hinblick auf deren Relevanz für die spätere Evaluation.

Zunächst listet Tabelle 5.1.4.1 die verwendeten Programme auf. Die Liste ist hierbei nach der Größe der zugehörigen IML-Dateien sortiert. Wir geben dabei neben der Versionsnummer auch den Programmnamen an. Um zu zeigen, dass wir uns bei der Auswahl der Programme nicht auf ein bestimmtes Anwendungsgebiet beschränkt haben, haben wir noch eine Kurzbeschreibung hinzugefügt.

Alle Programme sind aus dem Testrepository von BAUHAUS in der Sprache C. Da sich das Erzeugen der „.iml“-Dateien für die Programme als aufwändig herausgestellt hat, wurden für diese Arbeit bereits generierte „.iml“-Dateien zur Verfügung gestellt.
Aus diesen „.iml“-Dateien habe ich jene ausgewählt, welche die Analysekette von BAUHAUS bis zum thread_tool korrekt durchlaufen.

Die Schritte von der „.iml“-Datei bis zur „.tt“-Datei wurden dann, wie in Abschnitt 2.1 beschrieben, selbst durchgeführt.
Die weiteren Tabellen und Diagramme in diesem Abschnitt beschreiben die Größe der analysierten Programme mit verschiedenen Metriken.

Programm	Version	Beschreibung
Time.tt	1.7	Werkzeug zur Messung der Laufzeit
Mkeyes.tt	3.6	Pattern-Compiler für die eyeshape Datenbank
Darkhttpd.tt	1.2	Einfacher Webserver
dc.tt	1.06	Kommandozeilenrechner
Concepts.tt	0.3f	
Joseki.tt	3.6	Joseki-Compiler
Gnuplot_x11	4.0.0	Kommandozeilenprogramm zur Darstellung von Messdaten
Bc.tt	1.06	Kommandozeilenrechner
Nano.tt	1.2.3	Editor
Gnugo.tt	3.6	Go-Spiel

Tabelle 5.1.4.1: Die analysierten Programme

Programm	Routines	BBs	calls
time.tt	25	283	140
Mkeyes.tt	4	166	47
Darkhttpd.tt	58	929	427
dc.tt	127	1.752	630
Concepts.tt	176	1.791	654
Joseki.tt	258	4.350	733
Gnuplot_x11	97	2.416	847
Bc.tt	149	2.848	1.037
Nano.tt	212	4.241	2.268
Gnugo.tt	2.989	47.242	11.961

Tabelle 5.1.4.2: Größe der analysierten Programme

Die Metriken in Tabelle 5.1.4.2 wurden mit den von BAUHAUS zur Verfügung gestellten Iteratoren für die entsprechenden Knotentypen erhoben.

Die Tabelle 5.1.4.2 zeigt die Anzahl der Routinen, der Grundblöcke und die Anzahl der Direct-Calls. Diese Metriken werden dann bei der Auswertung der Ergebnisse dazu verwendet zu prüfen, wie die Ansätze für die einzelnen Größen skalieren.

Wie der Tabelle 5.1.4.2 zu entnehmen ist, wurde darauf geachtet, eine möglichst gute Bandbreite an Größen abzudecken. Insbesondere in den nachfolgenden Grafiken ist zu erkennen, dass das Programm Gnugo dabei größenmäßig deutlich heraussticht. Es war mir jedoch wichtig, auch ein größeres Beispiel in der Auswahl dabei zu haben.

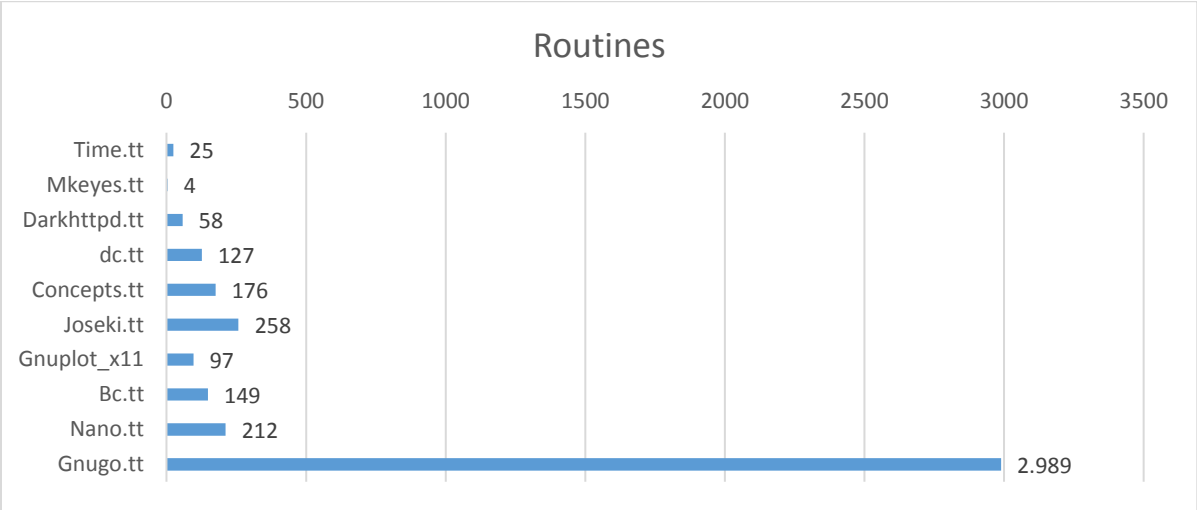

Abbildung 5.1.4.1: Anzahl Routinen

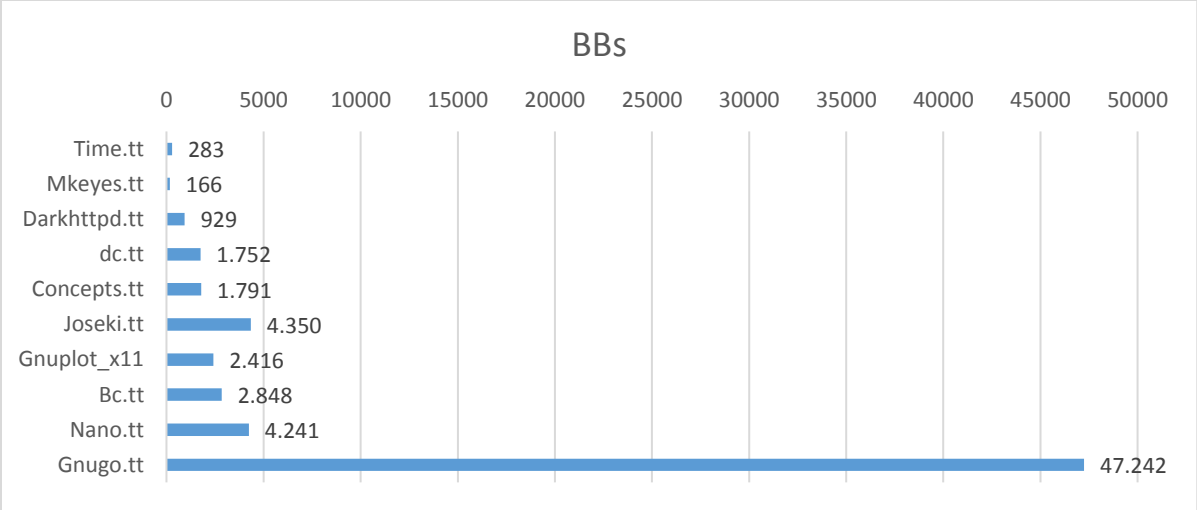

Abbildung 5.1.4.2: Anzahl Grundblöcke

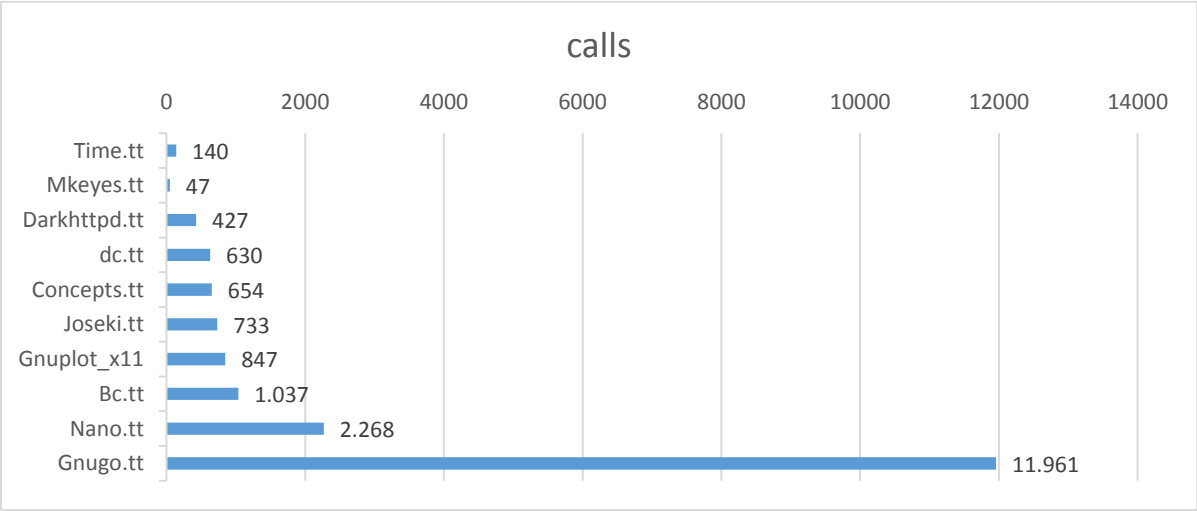

Abbildung 5.1.4.3: Anzahl Aufrufe

5.2. Messwerte

In diesem Abschnitt präsentieren wir die Resultate, welche wir im Rahmen des im vorigen Abschnitt beschriebenen Szenarios erhalten haben. Dieser Abschnitt stellt hierzu die Ergebnisse für vier verschiede Strategien mit Tabellen gegenüber. Die gezeigte Darstellung stellt eine Zusammenfassung da.

Da die Messwerte weit auseinander laufen, kam ein Mitteln über alle Werte nicht in Frage. Auch fanden sich, wie in den Grundlagen erwähnt, in anderen Arbeiten [2] Hinweise darauf, dass die Größe der Graphen einen Einfluss darauf haben kann, welcher Ansatz das bessere Laufzeitverhalten zeigt.
Ich entschied mich daher dazu, für diese Zusammenfassung der Ergebnisse, die Testprogramme in Gruppen einzuteilen. Die Einteilung der Gruppen erfolgte auf Basis der Anzahl der Grundblöcke. Für jede Gruppe wurde dann der Mittelwert gebildet und in einer Tabelle dargestellt. Diese Darstellung ermöglicht auch Aussagen darüber, wie die Ansätze für unterschiedlich große Graphen skalieren. Die exakten Messwerte sind in Anhang A zu finden.

5.2.1. Laufzeit

In diesem Unterabschnitt wird die Laufzeit der unterschiedlichen Ansätze betrachtet. Im intraprozeduralen Fall bleibt die Laufzeit für beide Datenflussprobleme erfreulich gering. Es fällt jedoch auf, dass die Implementierung von gültige Definitionen stellenweise schneller ist, als das vergleichbare Problem Lebendige Variablen. Besonders auffällig ist, dass Lebendige Variablen beim größten der untersuchten Programme fast doppelt so schnell ist, als das vergleichbare Problem Gültige Definitionen.

Intraprozedural - Gültige Definitionen

Anzahl BBs	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
< 500	25	20	10	10
≥ 500 & < 1.000	40	50	20	40
≥ 1.000 & < 2.000	110	90	115	120
≥ 2.000 & < 3.000	360	305	950	1.035
≥ 4.000 & < 5.000	380	345	535	655
≥ 40.000	442.320	442.910	238.650	242.380

Tabelle 5.2.1.1: Laufzeit gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural - Lebendige Variablen

Anzahl BBs	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
< 500	90	120	35	30
≥ 500 & < 1.000	40	40	50	40
≥ 1.000 & < 2.000	75	75	75	75
≥ 2.000 & < 3.000	280	280	290	285
≥ 4.000 & < 5.000	305	305	315	320
≥ 40.000	241.120	234.120	226.570	225.610

Tabelle 5.2.1.2: Laufzeit gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural – Postdominanz

Anzahl BBs	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
< 500	80	80	75	75
≥ 500 & < 1.000	150	150	140	140
≥ 1.000 & < 2.000	370	385	375	380
≥ 2.000 & < 3.000	1.370	1.380	1.385	1.370
≥ 4.000 & < 5.000	1.330	1.365	1.340	1.345
≥ 40.000	232.820	233.830	232.350	231.930

Tabelle 5.2.1.3: Laufzeit gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural - Dominanz

Anzahl BBs	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
< 500	80	80	80	80
≥ 500 & < 1.000	150	140	150	140
≥ 1.000 & < 2.000	385	385	385	385
≥ 2.000 & < 3.000	1.380	1.395	1.385	1.380
≥ 4.000 & < 5.000	1.325	1.360	1.335	1.350
≥ 40.000	228.990	229.190	240.250	232.840

Tabelle 5.2.1.4: Laufzeit gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Im interprozeduralen Fall zieht die Laufzeit bei beiden Datenflussproblemen deutlich an, bleibt jedoch immer noch in einem akzeptablen Bereich. Für das größte untersuchte Programm ist die interprozedurale Analyse sogar schneller als die intraprozedural Analyse. Nachforschungen nach der Ursache ergaben, dass bei der interprozeduralen Analyse offenbar nicht alle im Graphen vorhandenen Routinen besucht werden. Ursache dafür ist, dass es offenbar Routinen gibt, die nicht über einen Direct-Call erreichbar sind. Mögliche Erklärungen dafür sind toter Code oder das Vorhandensein mehrerer main-Methoden.

Interprozedural - Gültige Definitionen

Anzahl BBs	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
< 500	20	15	15	10
≥ 500 & < 1.000	70	70	50	40
≥ 1.000 & < 2.000	280	230	235	235
≥ 2.000 & < 3.000	805	810	1.530	1.610
≥ 4.000 & < 5.000	565	375	435	650
≥ 40.000	28.320	22.660	13.870	16.260

Tabelle 5.2.1.5: Laufzeit gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Interprozedural - Lebendige Variablen

Anzahl BBs	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
< 500	55	30	25	20
≥ 500 & < 1.000	120	110	50	40
≥ 1.000 & < 2.000	780	435	255	260
≥ 2.000 & < 3.000	3.105	1.160	580	610
≥ 4.000 & < 5.000	305	305	315	320
≥ 40.000	52.840	23.210	23.260	23.690

Tabelle 5.2.1.6: Laufzeit gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

5.2.2. Speicherverbrauch

In diesem Unterabschnitt wird der Speicherverbrauch der unterschiedlichen Ansätze betrachtet.

Es fällt auf, dass die Implementierung von Lebendige Variablen weniger Speicher verbraucht als die Implementierung von Gültige Definitionen. Ursache dafür ist, dass Lebendige Variablen in der Kill-Menge nur O_Variablen-Knoten speichert, während Gültige Definitionen Entity_L_Value-Knoten in der Def-Menge hat.

Intraprozedural - Gültige Definitionen

Anzahl BBs	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
< 500	<1	<1	<1	<1
≥ 500 & < 1.000	<1	<1	<1	<1
≥ 1.000 & < 2.000	6,5	5,5	9,5	11,5
≥ 2.000 & < 3.000	4,5	4,0	< 9	9,0
≥ 4.000 & < 5.000	6,5	5,5	9,5	11,5
≥ 40.000	126,0	108,0	137,0	173,0

Tabelle 5.2.2.1: Speicherverbrauch gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe.

Intraprozedural - Lebendige Variablen

Anzahl BBs	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
< 500	< 1,0	< 1,0	< 1,0	< 1,0
≥ 500 & < 1.000	1,0	1,0	1,0	1,0
≥ 1.000 & < 2.000	1,0	1,0	1,0	1,0
≥ 2.000 & < 3.000	3,0	2,5	3,0	3,0
≥ 4.000 & < 5.000	5,5	5,5	5,5	5,5
≥ 40.000	817,0	740,0	261,0	209,0

Tabelle 5.2.2.2: Speicherverbrauch gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe.

Intraprozedural – Postdominanz

Anzahl BBs	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
< 500	35,5	35,5	35,5	35,5
≥ 500 & < 1.000	1,0	1,0	1,0	1,0
≥ 1.000 & < 2.000	1,0	1,0	1,0	1,0
≥ 2.000 & < 3.000	1,0	1,0	1,0	1,0
≥ 4.000 & < 5.000	1,0	1,0	1,0	1,0
≥ 40.000	7,0	7,0	6,0	6,0

Tabelle 5.2.2.3: Speicherverbrauch gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe.

Intraprozedural – Dominanz

Anzahl BBs	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
< 500	35	35	35	35
≥ 500 & < 1.000	<1	<1	<1	<1
≥ 1.000 & < 2.000	<1	<1	<1	<1
≥ 2.000 & < 3.000	<1	<1	<1	<1
≥ 4.000 & < 5.000	<1	<1	<1	<1
≥ 40.000	9	10	10	9

Tabelle 5.2.2.4: Speicherverbrauch gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe.

Für die interprozedurale Analyse steigt der Speicherverbrauch merklich an, bleibt aber noch im Rahmen. Dies ist dadurch zu erklären, dass die Mengen an den Connectoren sehr groß werden können.

Interprozedural - Gültige Definitionen

Anzahl BBs	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
< 500	< 1,0	< 1,0	< 1,0	< 1,0
≥ 500 & < 1.000	< 1,0	1,0	< 1,0	< 1,0
≥ 1.000 & < 2.000	4,5	3,5	3,5	3,5
≥ 2.000 & < 3.000	14,0	14,0	20,5	21,0
≥ 4.000 & < 5.000	10,0	6,0	7,0	9,0
≥ 40.000	514,0	418,0	231,0	274,0

Tabelle 5.2.2.5: Speicherverbrauch gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe.

Interprozedural - Lebendige Variablen

Anzahl BBs	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
< 500	2,0	1,0	< 1,0	< 1,0
≥ 500 & < 1.000	4,0	4,0	1,0	1,0
≥ 1.000 & < 2.000	32,5	17,5	9,5	9,0
≥ 2.000 & < 3.000	132,0	47,5	22,0	22,5
≥ 4.000 & < 5.000	40,5	26,0	9,5	10,0
≥ 40.000	1.635,0	735,0	735,0	737,0

Tabelle 5.2.2.6: Speicherverbrauch gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe.

5.2.3. Besuchsrate der Grundblöcke

Dieser Unterabschnitt betrachtet die Besuchsrate. Die Besuchsrate ist eine Größe, die nicht gemessen sondern errechnet wird. Da alle nötigen Daten für die Berechnung vorhanden sind, bietet es sich an, diese Metrik hinzuzunehmen. Die Besuchsrate errechnet sich, indem man die Zahl der besuchten Grundblöcke durch die Zahl aller Grundblöcke teilt.
Die Besuchsrate bleibt in den meisten Fällen im einstelligen Bereich.

Intraprozedural - Gültige Definitionen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	4,50	4,85	1,90	1,95
≥ 500 & < 1.000	3,70	3,18	1,77	1,92
≥ 1.000 & < 2.000	3,47	2,77	1,92	2,17
≥ 2.000 & < 3.000	4,19	2,88	3,19	3,71
≥ 4.000 & < 5.000	3,76	3,01	2,53	3,54
≥ 40.000	4,31	3,73	2,38	3,09

Tabelle 5.2.3.1: Besuchsrate gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural - Lebendige Variablen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	6,85	7,10	2,78	2,39
≥ 500 & < 1.000	4,90	3,81	1,63	1,60
≥ 1.000 & < 2.000	6,27	3,02	1,56	1,57
≥ 2.000 & < 3.000	13,15	3,43	1,57	1,61
≥ 4.000 & < 5.000	5,83	3,64	1,72	1,82
≥ 40.000	5,93	4,52	2,26	2,10

Tabelle 5.2.3.2: Besuchsrate gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural - Postdominanz

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	7,73	7,66	3,27	1,75
≥ 500 & < 1.000	3,22	4,05	1,74	1,49
≥ 1.000 & < 2.000	4,66	3,27	1,54	1,56
≥ 2.000 & < 3.000	3,75	3,66	1,83	1,57
≥ 4.000 & < 5.000	4,20	3,86	1,96	1,67
≥ 40.000	4,25	5,04	3,31	1,66

Tabelle 5.2.3.3: Besuchsrate gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural - Dominanz

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	4,01	4,25	3,67	3,96
≥ 500 & < 1.000	3,52	3,00	1,56	3,10
≥ 1.000 & < 2.000	2,99	2,74	1,26	2,20
≥ 2.000 & < 3.000	3,68	2,84	1,41	3,01
≥ 4.000 & < 5.000	5,83	3,64	1,72	1,82
≥ 40.000	5,93	4,52	2,26	2,10

Tabelle 5.2.3.4: Besuchsrate gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Interprozedural - Gültige Definitionen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	4,99	5,17	1,98	2,09
≥ 500 & < 1.000	4,72	4,29	2,43	2,54
≥ 1.000 & < 2.000	5,26	4,34	3,53	3,86
≥ 2.000 & < 3.000	6,39	4,86	4,14	4,99
≥ 4.000 & < 5.000	3,20	2,41	2,29	3,03
≥ 40.000	6,60	5,38	3,17	3,89

Tabelle 5.2.3.5: Besuchsrate gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Interprozedural - Lebendige Variablen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	7,96	7,16	3,05	2,62
≥ 500 & < 1.000	5,76	4,70	2,24	2,25
≥ 1.000 & < 2.000	8,42	4,29	2,96	2,99
≥ 2.000 & < 3.000	16,72	5,31	3,26	3,51
≥ 4.000 & < 5.000	4,35	2,90	1,49	1,62
≥ 40.000	8,08	6,89	4,01	4,03

Tabelle 5.2.3.6: Besuchsrate gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

5.2.4. Anzahl besuchter Grundblöcke

Dieser Unterabschnitt befasst sich mit der Anzahl der besuchten Grundblöcke.

Anhand der nachfolgenden Tabellen ist ersichtlich, dass die Grundblöcke von den Worklist-Implementierungen am wenigsten besucht werden. Es fällt auf, dass die Worklist-Implementierung, die topologisch sortiert ist, stellenweise mehr Grundblöcke besucht, als die Worklist-Implementierung mit Stack.

Die Häufigkeit mit der dieser Fall auftritt variiert nach Art des Datenflussproblems. Für das Datenflussproblem Gültige Definitionen tritt dieser Fall für den intraprozeduralen Fall in sechs von sechs Fällen auf. Im intraprozeduralen Fall des Datenflussproblems Lebendige Variablen tritt dieser Fall in drei von sechs Fällen auf. In nur einem von sechs Fällen kommt dieser Fall für das Datenflussproblem Postdominanz vor.
Da das Problem bei den Vorwärtsanalysen am häufigsten auftritt, habe ich bei den Untersuchungen ein besonderes Augenmerk auf die Vorwärtsanalysen gelegt. Ich habe nachgeprüft ob die Worklist in der richtigen Richtung durchlaufen wird und keine Duplikate enthält. Beide Punkte konnte ich als Ursachen für das Problem ausschließen, indem ich mir die Grundblöcke in der Reihenfolge in der sie besucht wurden, zur Überprüfung habe ausgeben lassen.

Um sicher zu gehen, dass die Worklist auch wirklich in topologischer Reihenfolge sortiert, habe ich die Worklist mit zwei unterschiedlichen Sortierkriterien getestet. Als Kriterien kamen sowohl die Postordernummer(in umgekehrter Reihenfolge) als auch die DFS-Nummer zum Einsatz. Die Sortierung nach der Postordernummer(in umgekehrter Reihenfolge) hat das Problem nur noch verschärft. Dies deutet darauf hin, dass die DFS-Nummer tatsächlich der topologischen Reihenfolge entspricht.
Eine weitere mögliche Fehlerquelle wäre noch, dass die in den Grundblöcken gespeicherten Nummern nicht immer korrekt sind. Ob diese Fehlerquelle zum Tragen kommt, konnte ich nicht ausschließen.

Intraprozedural - Gültige Definitionen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	883,0	954,5	395,0	408,5
≥ 500 & < 1.000	3.439,0	2.954,0	1.644,0	1.782,0
≥ 1.000 & < 2.000	6.145,0	4.912,5	3.407,0	3.862,0
≥ 2.000 & < 3.000	11.017,5	7.722,0	8.853,5	10.175,5
≥ 4.000 & < 5.000	16.116,5	12.879,5	10.782,5	15.148,0
≥ 40.000	203.576,0	176.321,0	112.355,0	145.894,0

Tabelle 5.2.4.1: Anzahl der besuchten Grundblöcke gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural - Lebendige Variablen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	1.493,5	1.387,5	554,5	484,5
≥ 500 & < 1.000	4.554,0	3.543,0	1.514,0	1.484,0
≥ 1.000 & < 2.000	11.061,0	5.359,0	2.775,5	2.789,5
≥ 2.000 & < 3.000	33.442,5	9.243,0	4.197,0	4.281,0
≥ 4.000 & < 5.000	24.997,5	15.608,5	7.386,0	7.830,5
≥ 40.000	280.218,0	213.720,0	106.534,0	99.148,0

Tabelle 5.2.4.2: Anzahl der besuchten Grundblöcke gemittelt über alle Graphen der Testprogramme aus der entsprechenden Gruppe.

Intraprozedural – Postdominanz

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	1.339,0	1.492,5	656,0	379,0
≥ 500 & < 1.000	2.990,0	3.761,0	1.612,0	1.385,0
≥ 1.000 & < 2.000	8.227,5	5.802,5	2.728,0	2.765,5
≥ 2.000 & < 3.000	9.867,5	9.878,5	4.949,5	4.162,0
≥ 4.000 & < 5.000	18.011,5	16.566,5	8.405,0	7.177,5
≥ 40.000	200.567,0	237.932,0	156.569,0	78.460,0

Tabelle 5.2.4.3: Anzahl besuchte BBs gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe

Intraprozedural – Dominanz

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	853,0	880,0	683,0	768,0
≥ 500 & < 1.000	3.273,0	2.787,0	1.449,0	2.876,0
≥ 1.000 & < 2.000	5.293,5	4.852,5	2.232,0	3.895,0
≥ 2.000 & < 3.000	9.654,0	7.597,0	3.763,0	7.850,5
≥ 4.000 & < 5.000	16.622,0	12.796,0	8.378,5	25.038,5
≥ 40.000	201.799,0	166.299,0	106.534,0	231.113,0

Tabelle 5.2.4.4: Anzahl besuchte BBs gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe

Interprozedural - Gültige Definitionen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	1.026,0	1.049,0	422,0	449,5
≥ 500 & < 1.000	4.383,0	3.988,0	2.256,0	2.359,0
≥ 1.000 & < 2.000	9.308,0	7.676,5	6.254,0	6.845,0
≥ 2.000 & < 3.000	16.771,0	12.978,5	11.144,0	13.309,5
≥ 4.000 & < 5.000	13.683,0	10.299,5	9.755,5	12.911,5
≥ 40.000	311.914,0	254.038,0	149.756,0	183.962,0

Tabelle 5.2.4.5 Anzahl besuchte BBs gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe

Interprozedural - Lebendige Variablen

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	1.809,0	1.409,0	634,0	554,0
≥ 500 & < 1.000	5.355,0	4.363,0	2.085,0	2.086,0
≥ 1.000 & < 2.000	14.862,5	7.598,5	5.247,5	5.302,0
≥ 2.000 & < 3.000	42.753,0	14.191,5	8.716,5	9.379,0
≥ 4.000 & < 5.000	18.566,5	12.398,5	6.251,5	6.910,5
≥ 40.000	381.612,0	325.501,0	189.400,0	190.325,0

Tabelle 5.2.4.6: Anzahl besuchte BBs gemittelt über alle Graphen der Testprogramme aus der entspr. Gruppe

5.3. Vergleich

Dieser Abschnitt vergleicht die im vorigen Abschnitt gezeigten Resultate mit Hilfe von Diagrammen und diskutiert auf Basis dieser die Auswirkungen der unterschiedlichen Ansätze. Der Vergleich der Ansätze erfolgt für den intraprozeduralen Fall und den interprozeduralen Fall jeweils getrennt in separaten Unterabschnitten.

Da die größte Gruppe die Diagramme stark streckt, wurde jeweils noch ein Diagramm ohne die größte Gruppe eingefügt. Der Vergleich findet für den intraprozeduralen Fall und den interprozeduralen Fall getrennt statt. Beide Fälle werden in einem separaten Abschnitt abgehandelt. Im Anschluss werden die beiden Fälle dann noch verglichen.

5.3.1. Intraprozedural – Gültige Definitionen

Dieser Unterabschnitt befasst sich mit dem Fall der intraprozeduralen Analyse für Gültige Definitionen.

Die Diagramme veranschaulichen, welcher Ansatz für welche Gruppe am besten abschneidet.

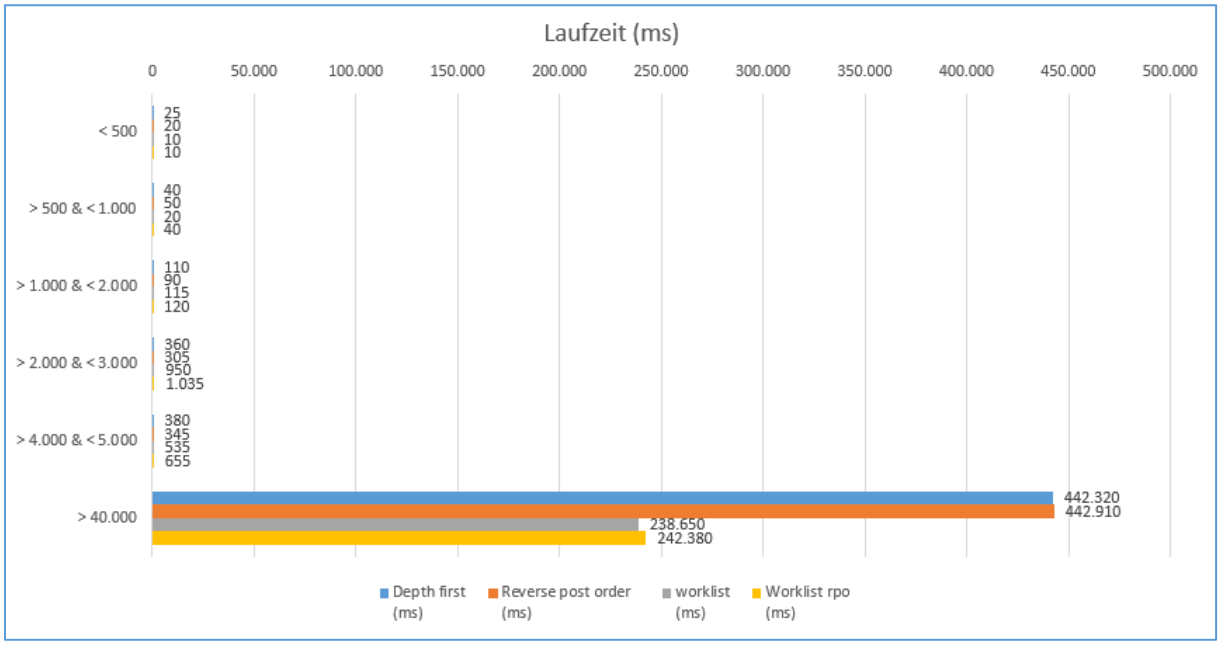

Abbildung 5.3.1.1: Laufzeit für alle Gruppen

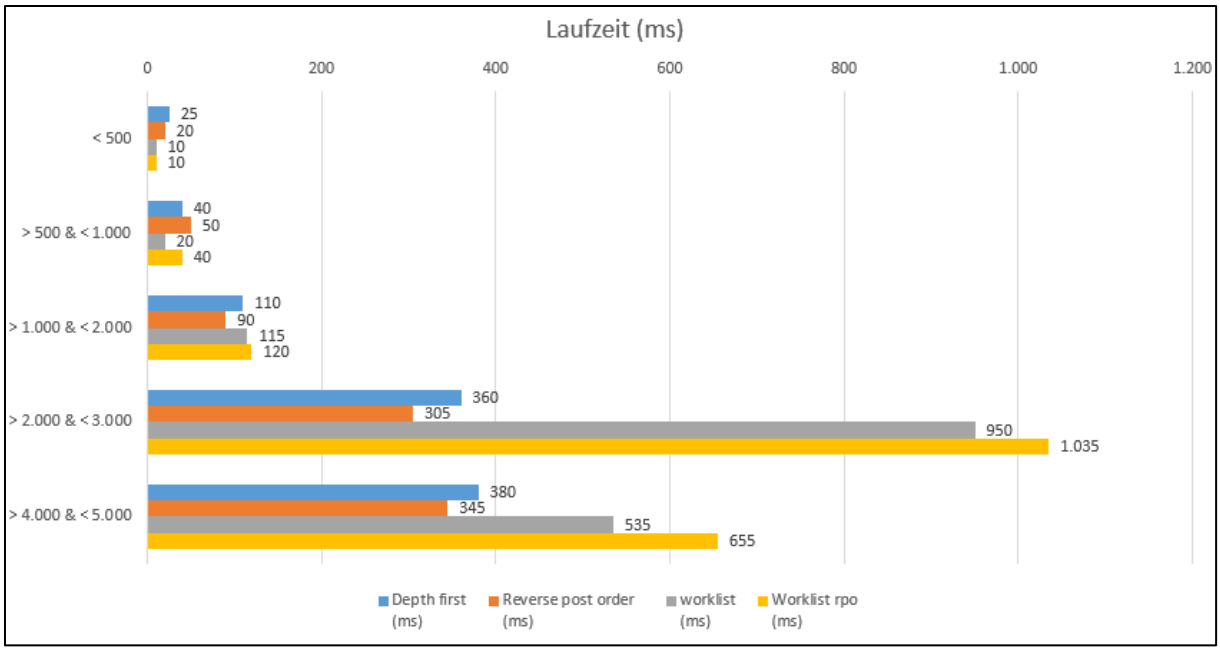

Abbildung 5.3.1.2: Laufzeit ohne größte Gruppe

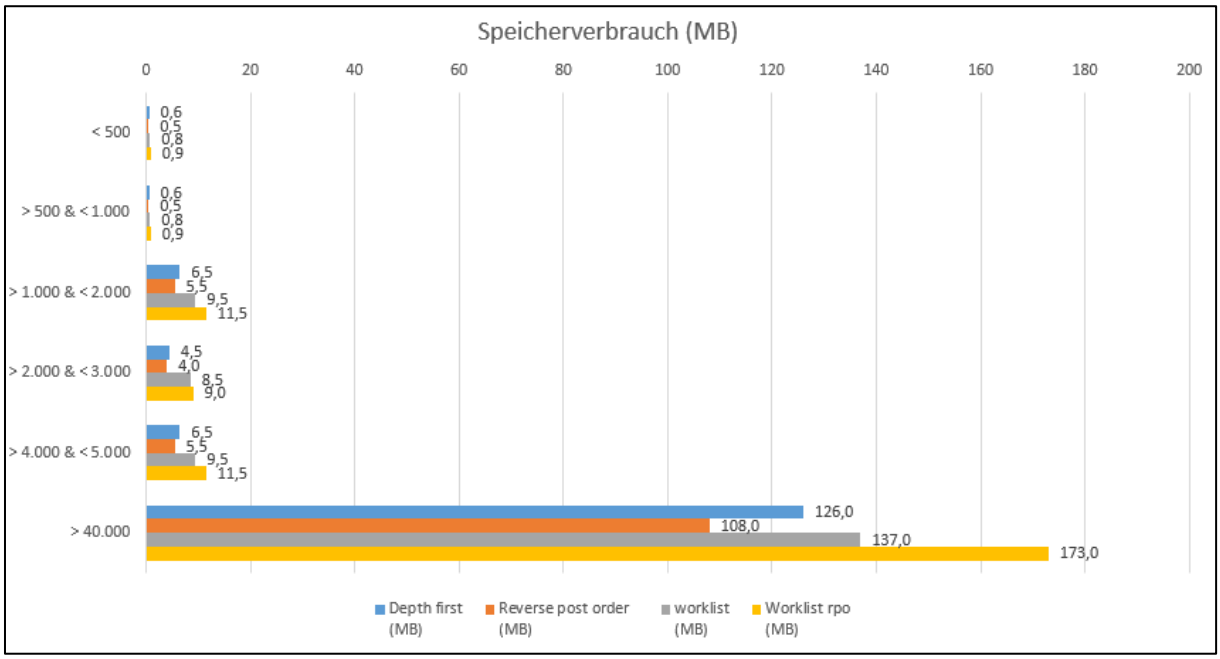

Abbildung 5.3.1.3: Speicherverbrauch für alle Gruppen

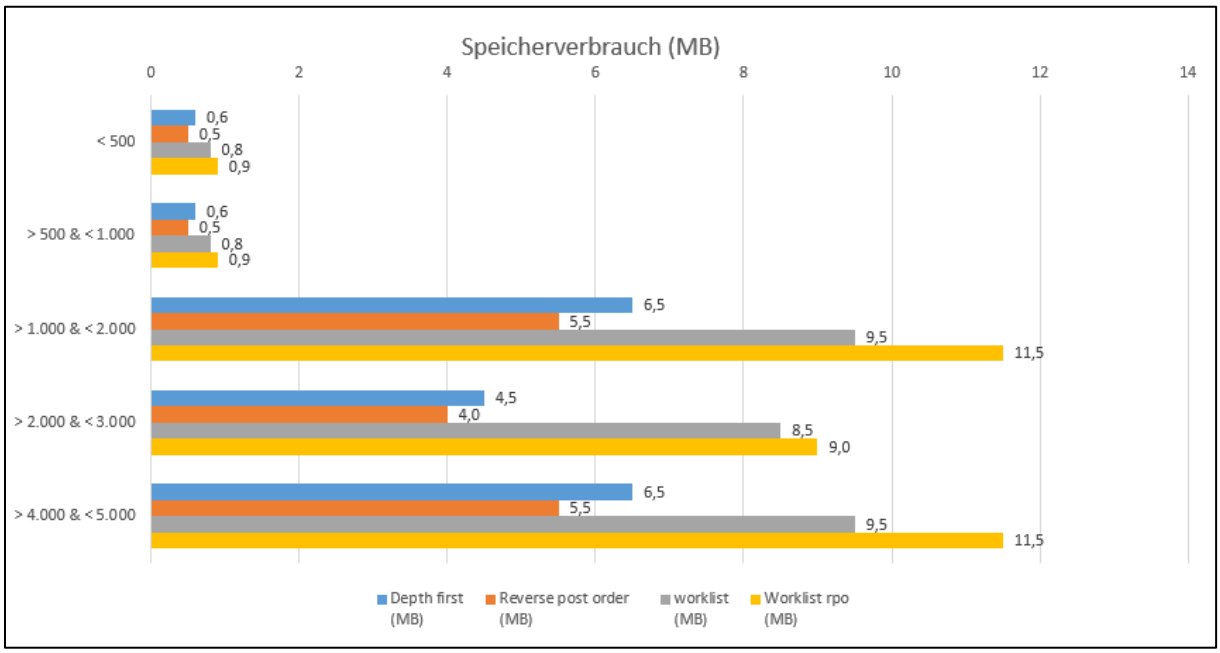

Abbildung 5.3.1.4: Speicherverbrauch ohne größte Gruppe

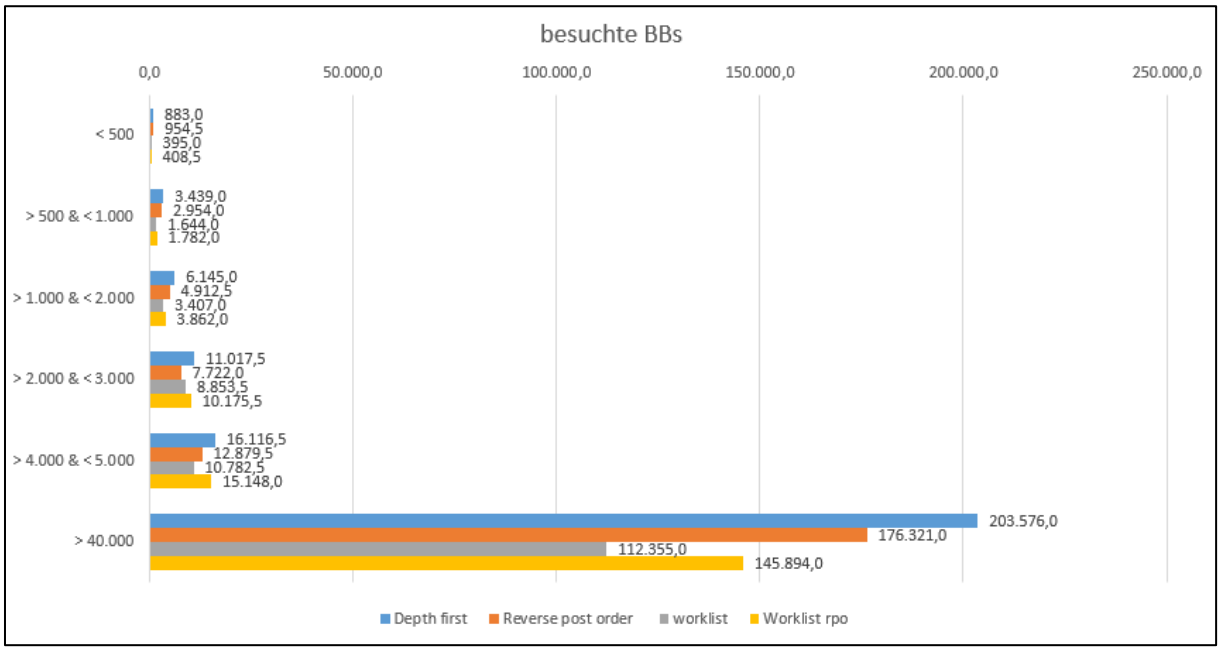

Abbildung 5.3.1.5: Anzahl besuchte Grundblöcke für alle Gruppen

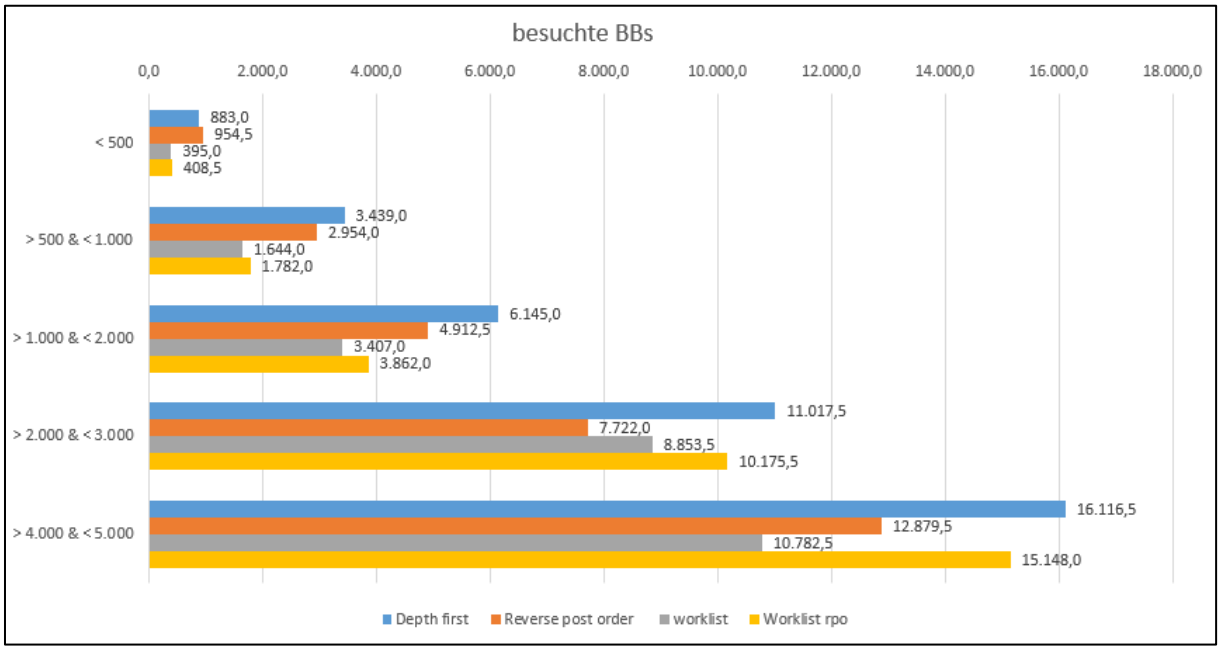

Abbildung 5.3.1.6: Anzahl besuchte Grundblöcke ohne größte Gruppe

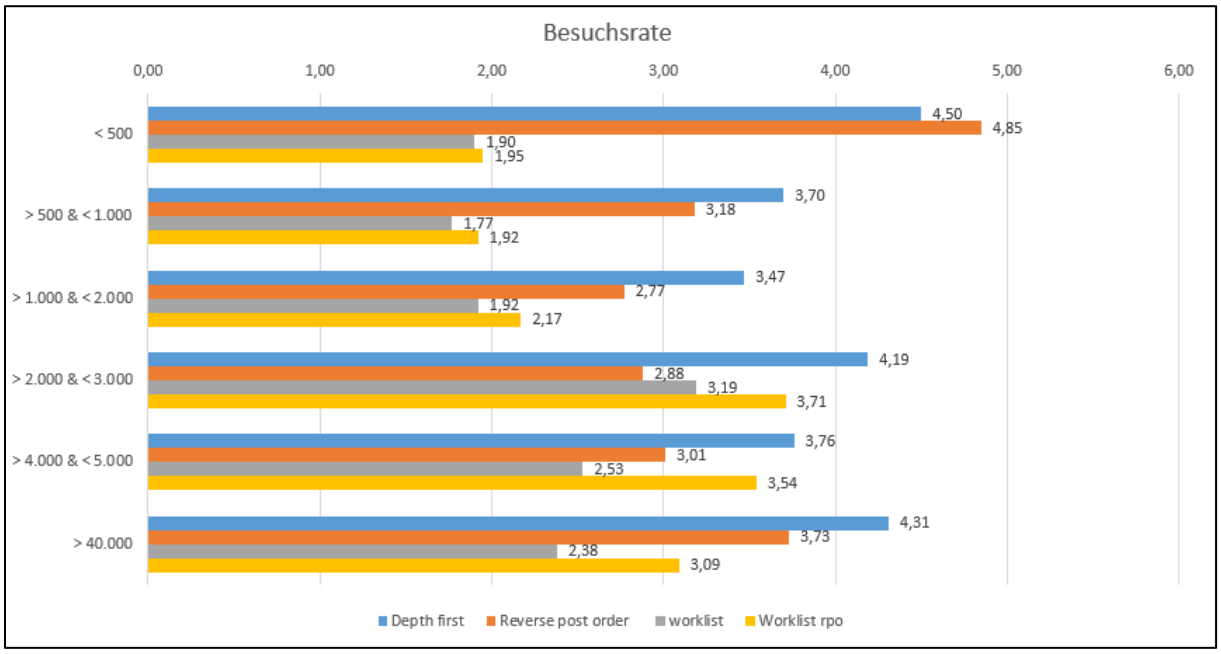

Abbildung 5.3.1.7: Besuchsrate für alle Gruppen

Folgende Tabelle zeigt die Sieger aus der jeweiligen Gruppe:

Anzahl BBs	Laufzeit	Speicher-verbrauch	Besuchsrate	BBs
< 500	worklist/worklist rpo	RPO	worklist	worklist
≥ 500 & < 1.000	worklist	RPO	worklist	worklist
≥ 1.000 & < 2.000	RPO	RPO	worklist	worklist
≥ 2.000 & < 3.000	RPO	RPO	RPO	RPO
≥ 4.000 & < 5.000	RPO	RPO	worklist	worklist
≥ 40.000	worklist	RPO	worklist	worklist

Tabelle 5.3.1.1: Sieger der jeweiligen Gruppe

Wie man sieht, wechseln sich in der Kategorie Laufzeit die Ansätze Worklist und RPO (reverse Postorder) als Sieger ab. Der Worklist-Ansatz schneidet sowohl für kleine als auch sehr große Graphen am besten ab. Im Mittelfeld dagegen macht sich der RPO-Ansatz besser. In der Kategorie Speicherverbrauch schneidet der RPO-Ansatz für alle Gruppen am besten ab.
Bei der Anzahl der besuchten Grundblöcke und der Besuchsrate ist der Worklist-Ansatz in fünf von sechs Fällen am besten.

Um nun zu einer Entscheidung zu kommen welcher Ansatz insgesamt am besten ist, brauchen wir eine Wertung. Vergeben wir nun für jeden Kategorie Sieg einen Punkt kommen wir zu folgender Wertung:

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	0	1	3	1
≥ 500 & < 1.000	0	1	3	0
≥ 1.000 & < 2.000	0	2	2	0
≥ 2.000 & < 3.000	0	4	0	0
≥ 4.000 & < 5.000	0	2	2	0
≥ 40.000	0	1	3	0

Tabelle 5.3.1.2: Wertung nach Gruppe

Wie man sieht, schneidet der Worklist-Ansatz in den meisten Fällen am besten ab. An zweiter Stelle kommt der RPO-Ansatz. Die anderen beiden Ansätze konnten sich nicht positiv hervortun.

5.3.2. Intraprozedural – Lebendige Variablen

Dieser Unterabschnitt befasst sich mit dem Fall der intraprozeduralen Analyse für Lebendige Variablen.

Die Diagramme veranschaulichen, welcher Ansatz für welche Gruppe am besten abschneidet.

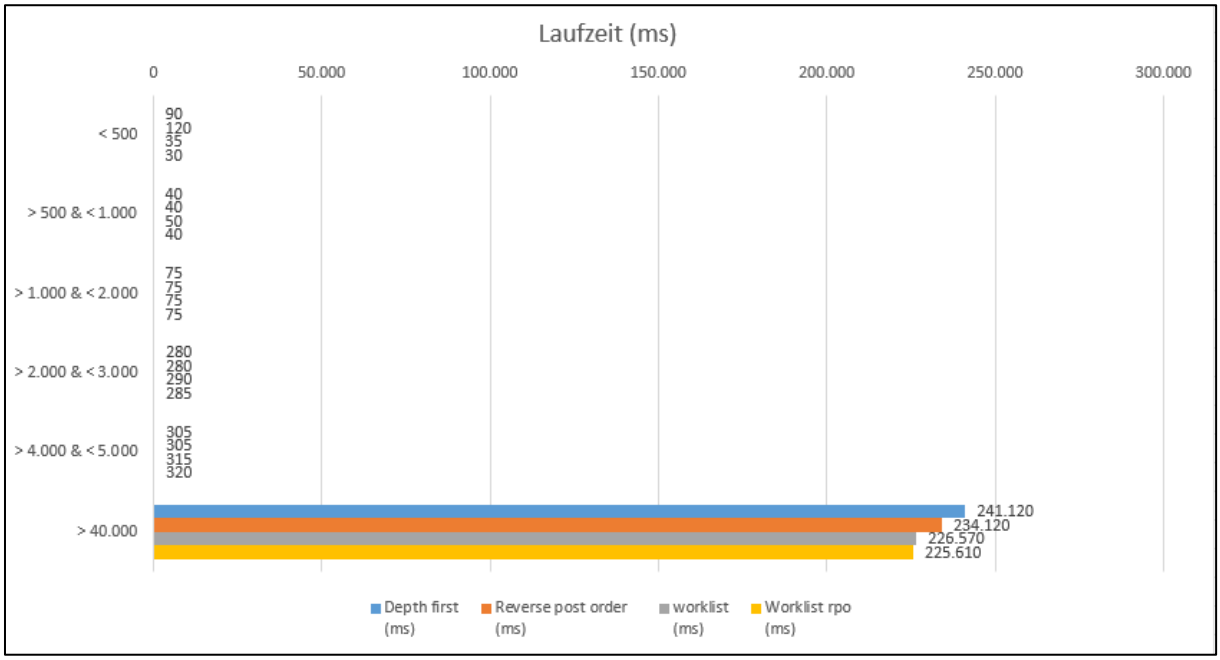

Abbildung 5.3.2.1: Laufzeit für alle Gruppen

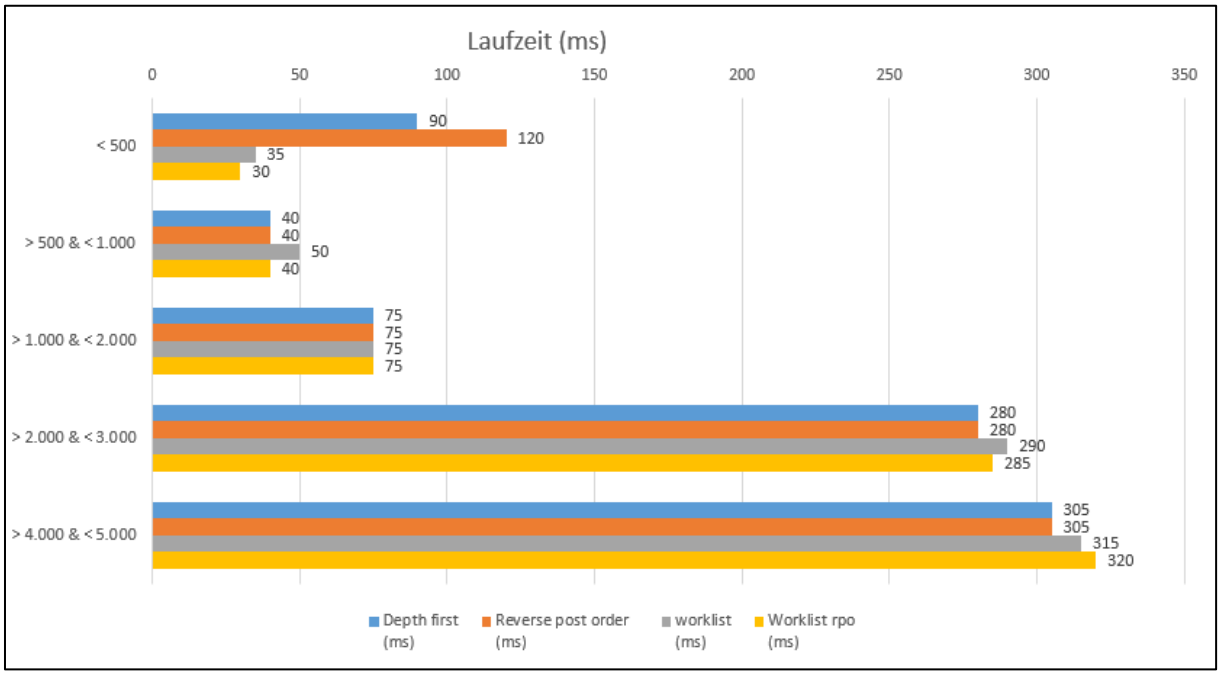

Abbildung 5.3.2.2: Laufzeit ohne größte Gruppe

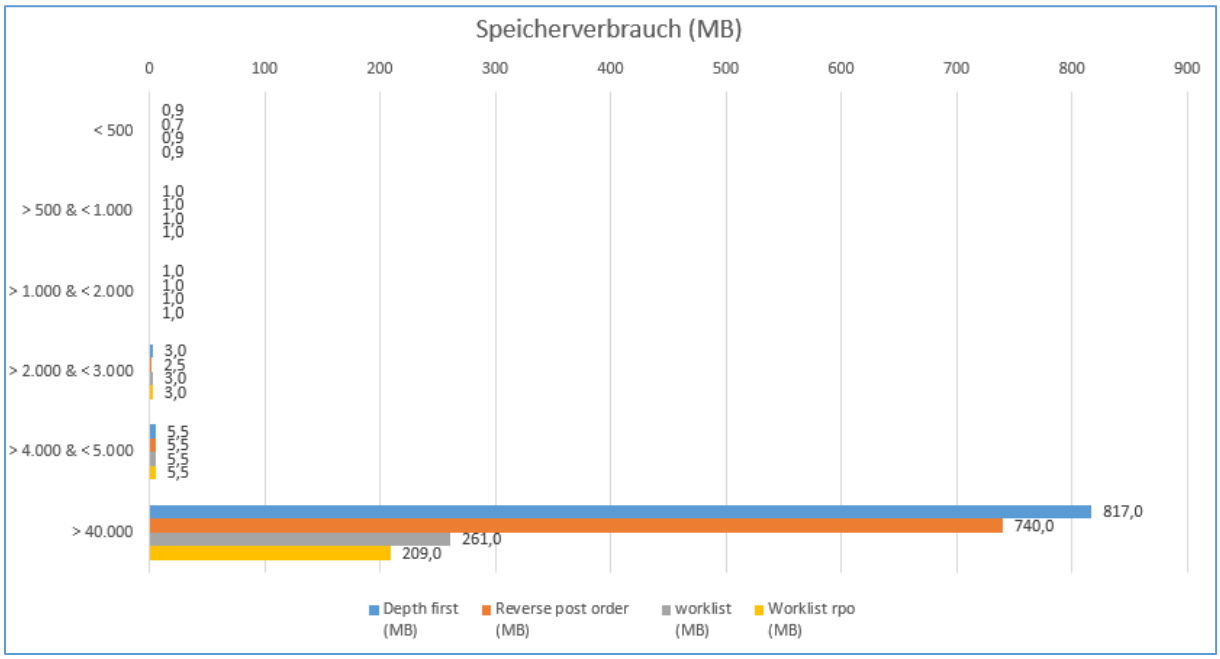

Abbildung 5.3.2.3: Speicherverbrauch alle Gruppen

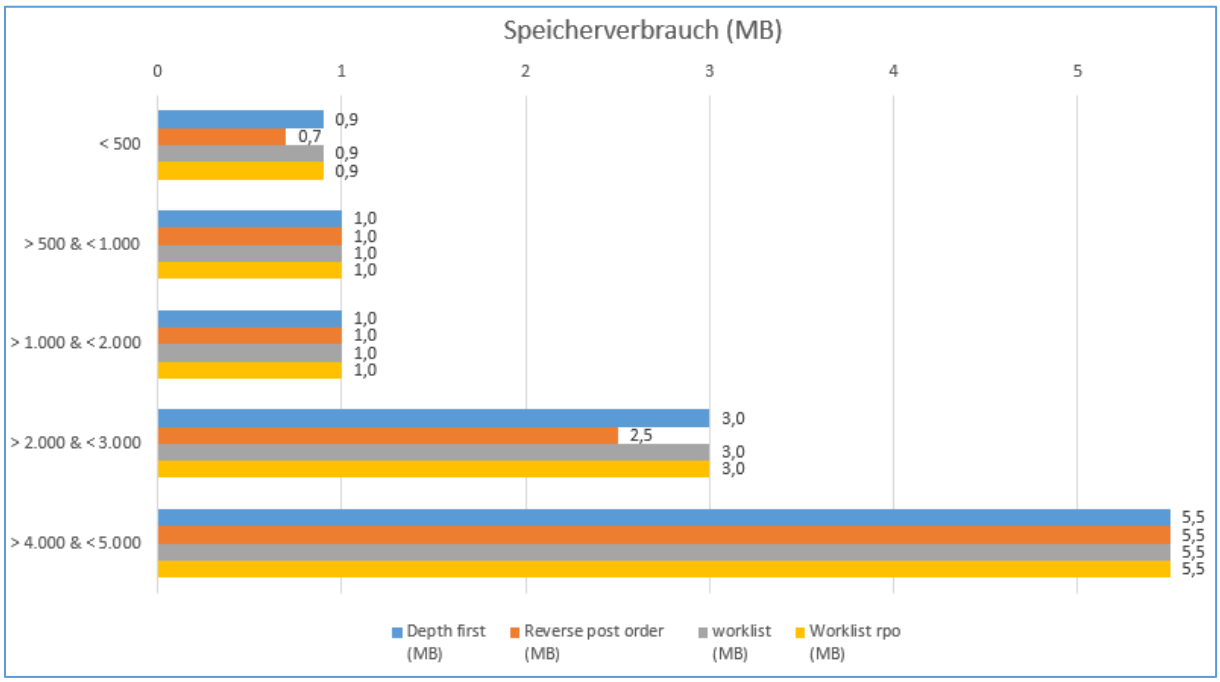

Abbildung 5.3.2.4: Speicherverbrauch ohne größte Gruppe

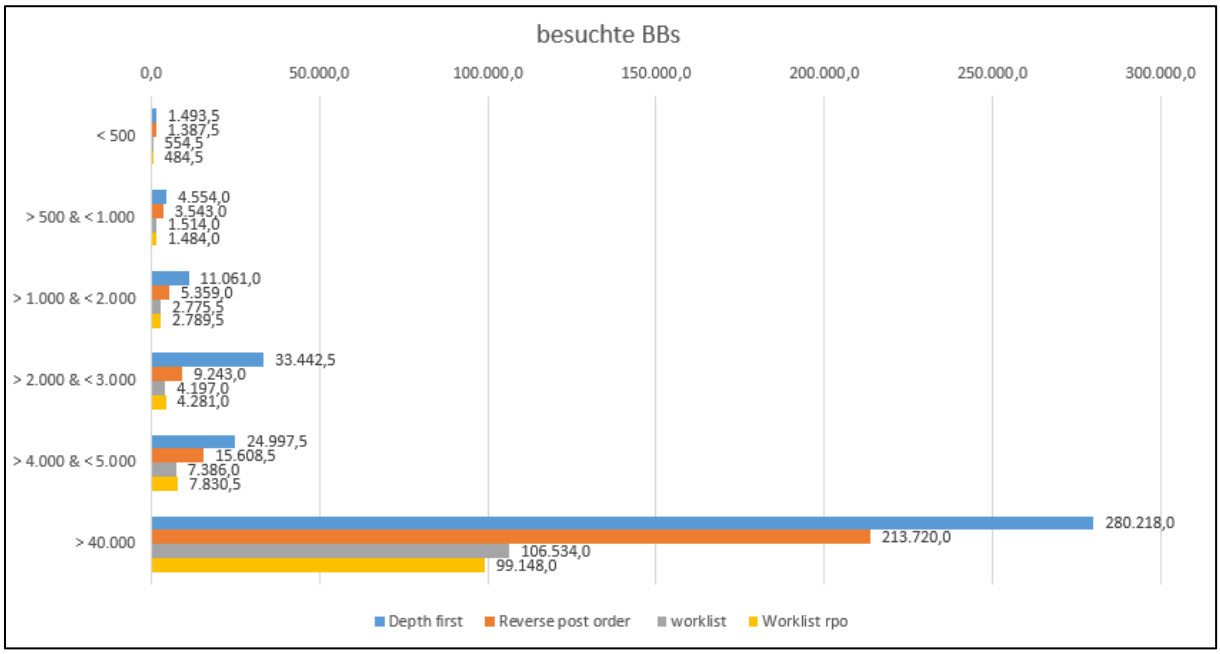

Abbildung 5.3.2.5: Anzahl besuchte Grundblöcke alle Gruppen

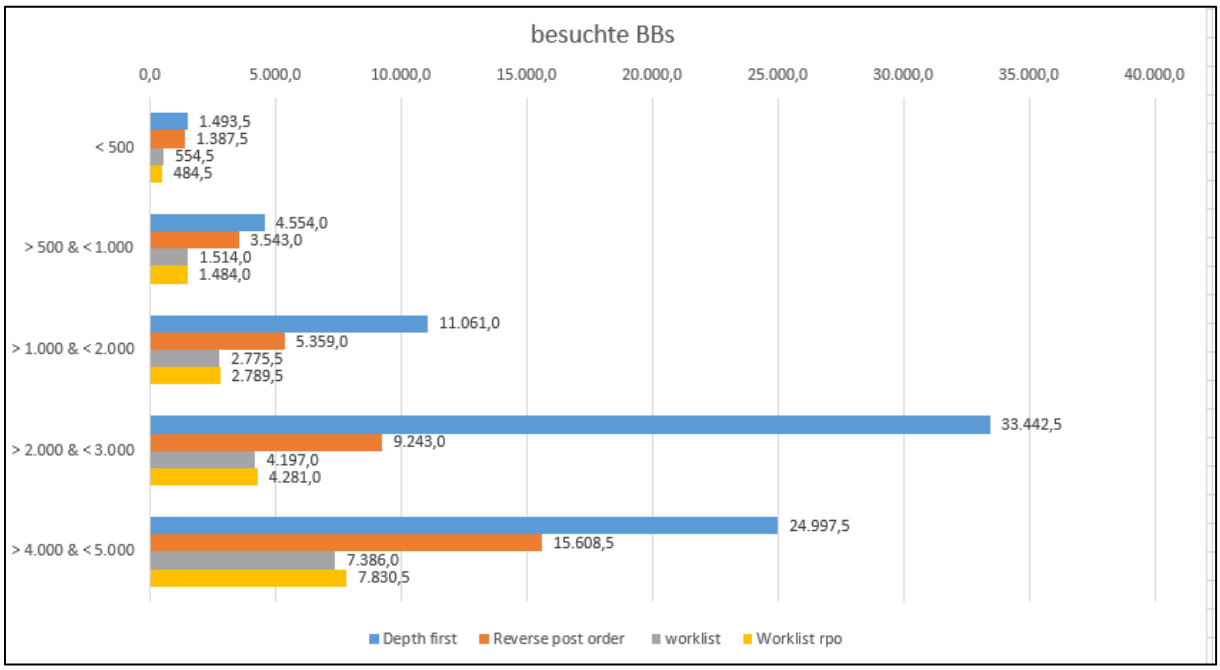

Abbildung 5.3.2.6: Anzahl besuchte Grundblöcke ohne größte Gruppe

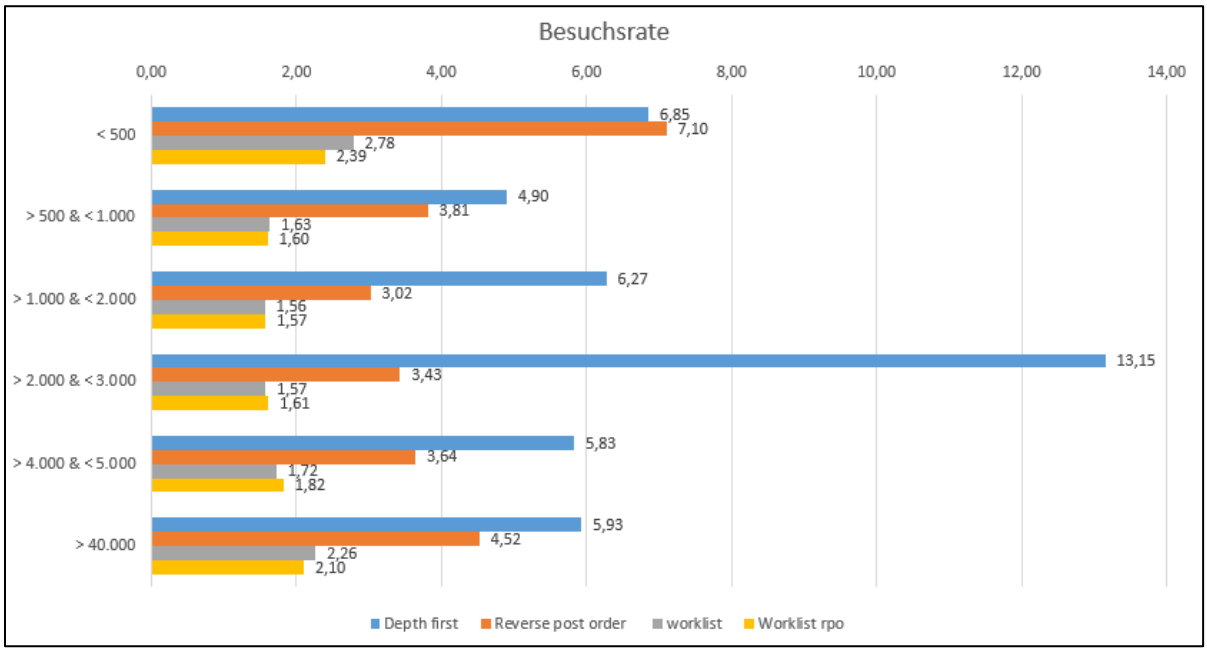

Abbildung 5.3.2.7: Besuchsrate alle Gruppen

Folgende Tabelle zeigt die Sieger aus der jeweiligen Gruppe:

Anzahl BBs	Laufzeit	Speicher-verbrauch	Besuchsrate	BBs
< 500	worklist rpo	RPO	worklist rpo	worklist rpo
≥ 500 & < 1.000	worklist rpo/RPO/DFO	alle	worklist rpo	worklist rpo
≥ 1.000 & < 2.000	alle	alle	worklist	worklist
≥ 2.000 & < 3.000	RPO/DFO	RPO	worklist	worklist
≥ 4.000 & < 5.000	RPO/DFO	alle	worklist	worklist
≥ 40.000	worklist rpo	worklist rpo	worklist rpo	worklist rpo

Tabelle 5.3.2.1: Sieger der jeweiligen Gruppe

Wie man sieht, wechseln sich in der Kategorie Laufzeit mehrere Ansätze als Sieger ab. Der Worklist rpo-Ansatz schneidet sowohl für sehr kleine, als auch sehr große Graphen am besten, bzw. gleich gut wie die anderen Ansätze ab. Im Mittelfeld dagegen macht sich die anderen Ansätze besser.
In der Kategorie Speicherverbrauch ist wieder der RPO-Ansatz am besten, bzw. zumindest gleich gut wie die anderen Ansätze. Bei der Besuchsrate i sowie der Anzahl der besuchten Blöcke sind die Ansätze Worklist und Worklist rpo nahezu identisch gut. Sowohl für die kleinsten als auch die größten der untersuchten Graphen ist der Worklist rpo Ansatz geringfügig besser als der Worklist Ansatz. Im Mittelfeld (1000 bis 5000 Grundblöcke) schneidet der Worklist Ansatz geringfügig besser ab.

Um nun zu einer Entscheidung zu kommen welcher Ansatz insgesamt am besten ist, brauchen wir eine Wertung. Vergeben wir nun für jeden Kategorie Sieg einen Punkt kommen wir zu folgender Wertung:

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	0	1	0	3
≥ 500 & < 1.000	2	2	1	4
≥ 1.000 & < 2.000	2	2	4	2
≥ 2.000 & < 3.000	1	2	2	0
≥ 4.000 & < 5.000	2	2	3	1
≥ 40.000	0	0	0	4

Tabelle 5.3.2.2: Wertung nach Gruppe

Wie man sieht schneidet der Worklist rpo-Ansatz in den meisten Fällen am besten ab. An zweiter Stelle liegen der Worklist- und der RPO-Ansatz. Den letzten Platz belegt der DFO-Ansatz.

Vergleichen wir nun die Wertungen der Analyse Gültige Definitionen mit denen der Analyse Lebendige Variablen.
Das Ergebnis fällt für Lebendige Variablen wesentlich differenzierter aus, alle Ansätze schneiden unterschiedlich gut ab. Insbesondere der Worklist rpo-Ansatz schneidet für Lebendige Variablen wesentlich besser ab als für Gültige Definitionen.

5.3.3. Intraprozedural – Postdominanz

Dieser Unterabschnitt befasst sich mit dem Fall der interprozeduralen Analyse für Postdominanz.

Die Diagramme veranschaulichen, welcher Ansatz für welche Gruppe am besten abschneidet.

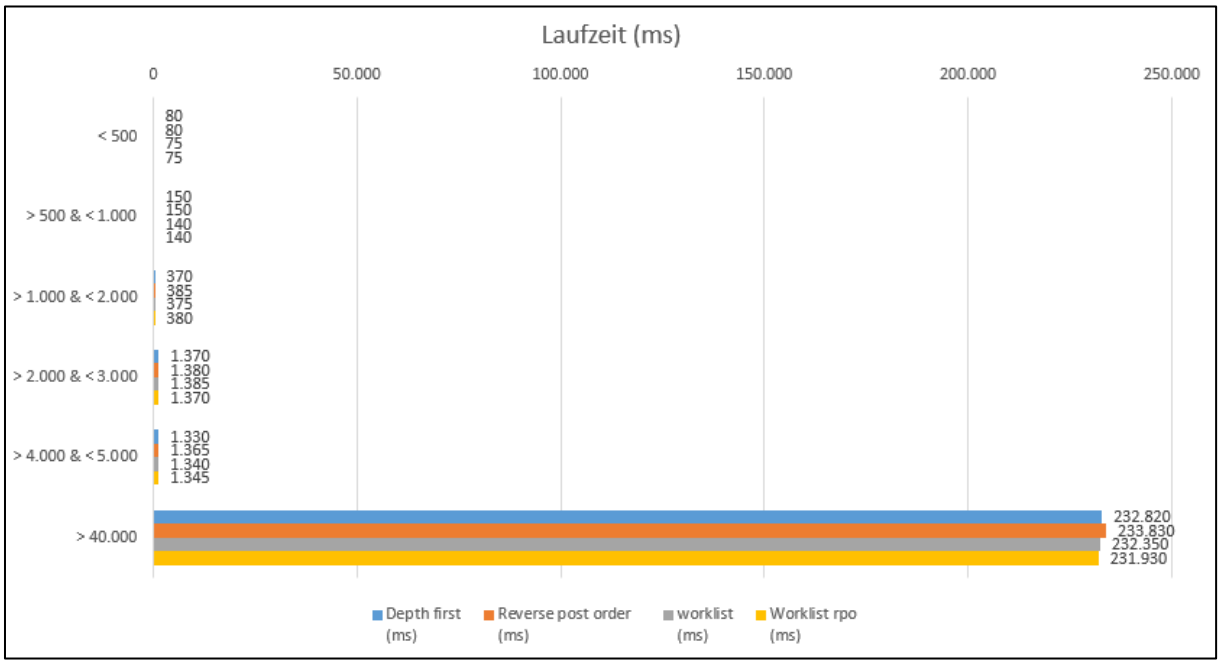

Abbildung 5.3.3.1: Laufzeit für alle Gruppen

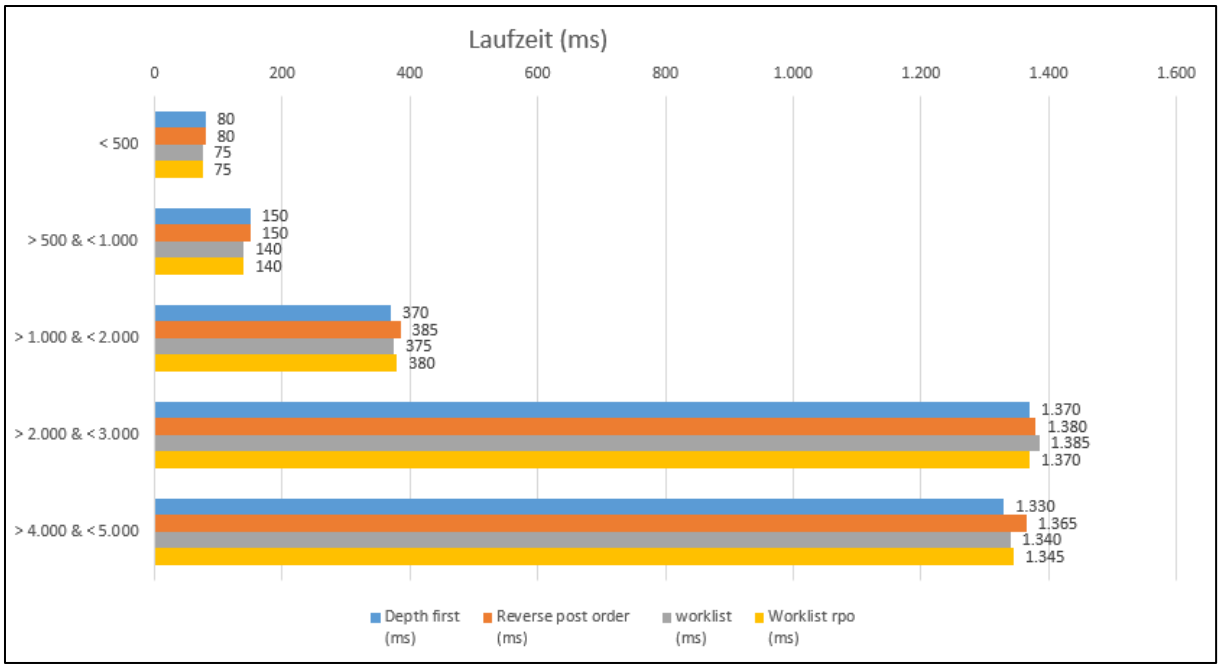

Abbildung 5.3.3.2: Laufzeit ohne größte Gruppe

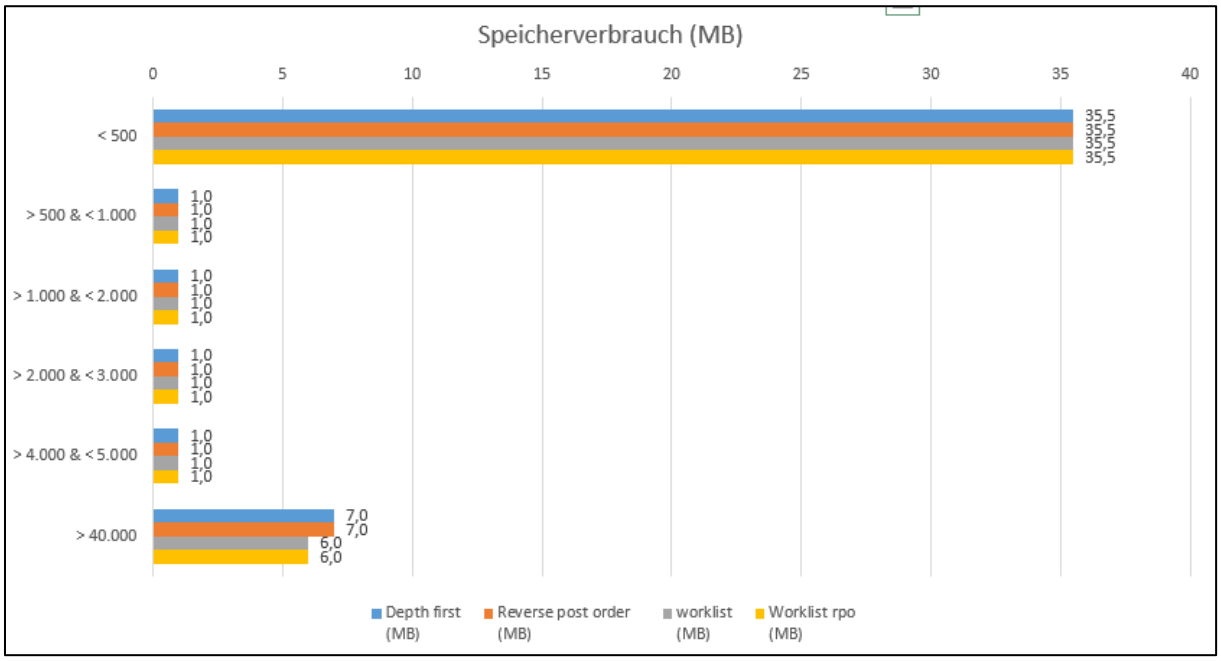

Abbildung 5.3.3.3: Speicherverbrauch alle Gruppen

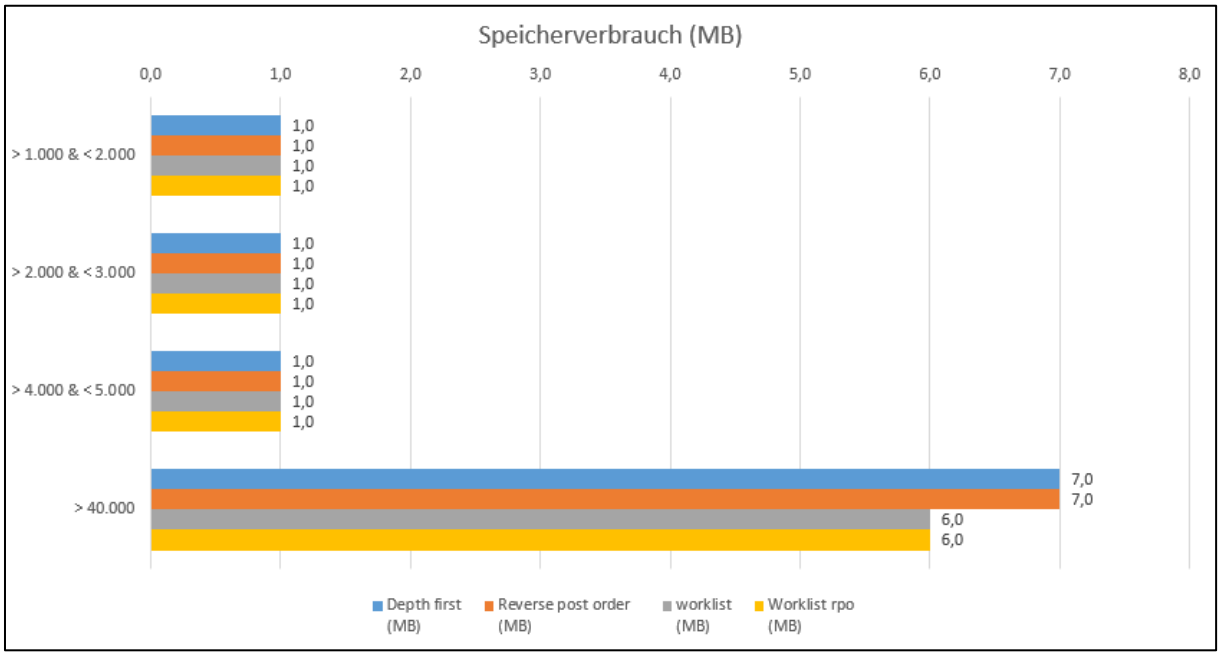

Abbildung 5.3.3.4: Speicherverbrauch ohne größte Gruppe

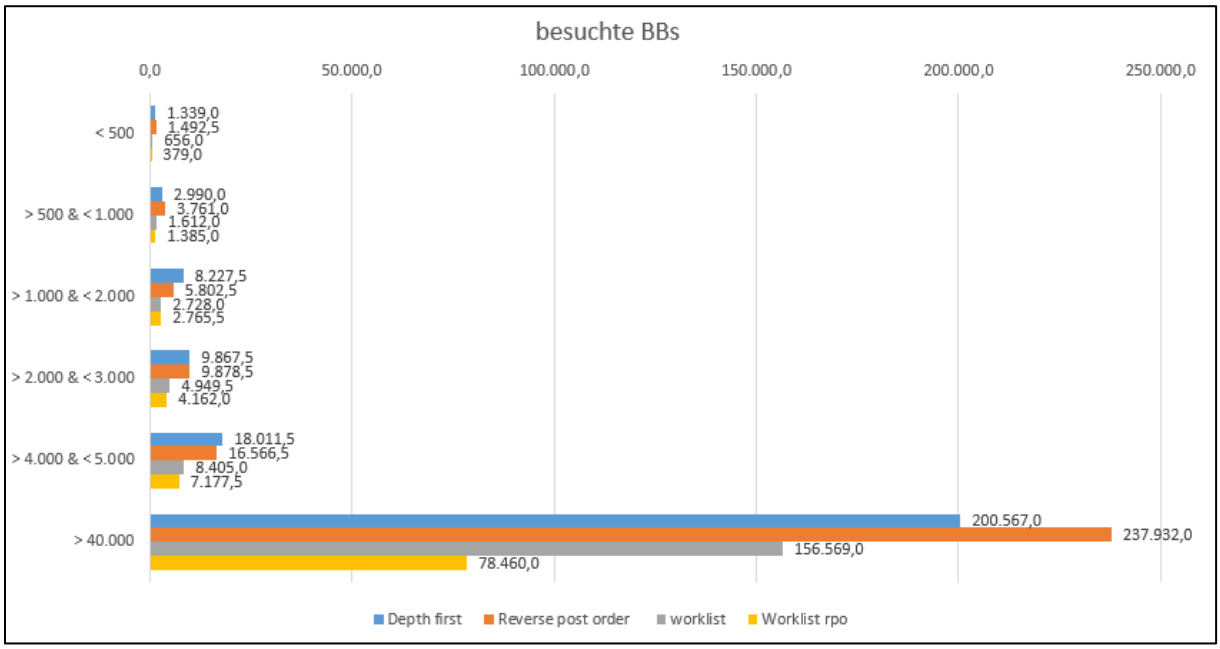

Abbildung 5.3.3.5: Anzahl besuchte Grundblöcke alle Gruppen

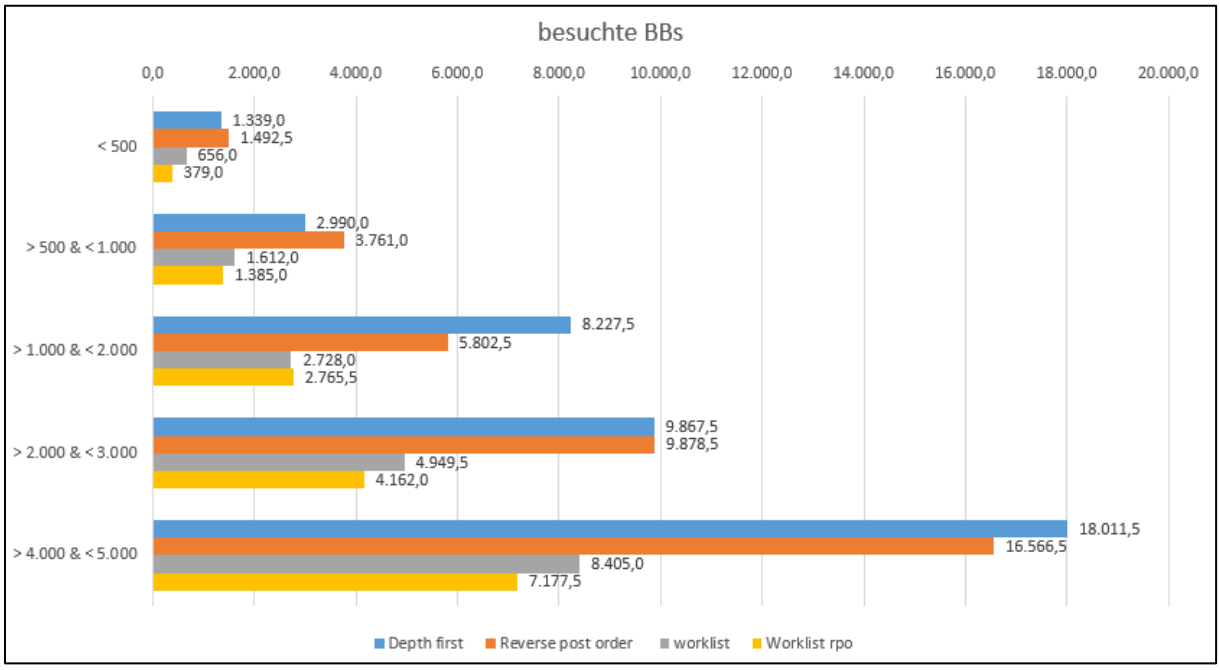

Abbildung 5.3.3.6: Anzahl besuchte Grundblöcke ohne größte Gruppe

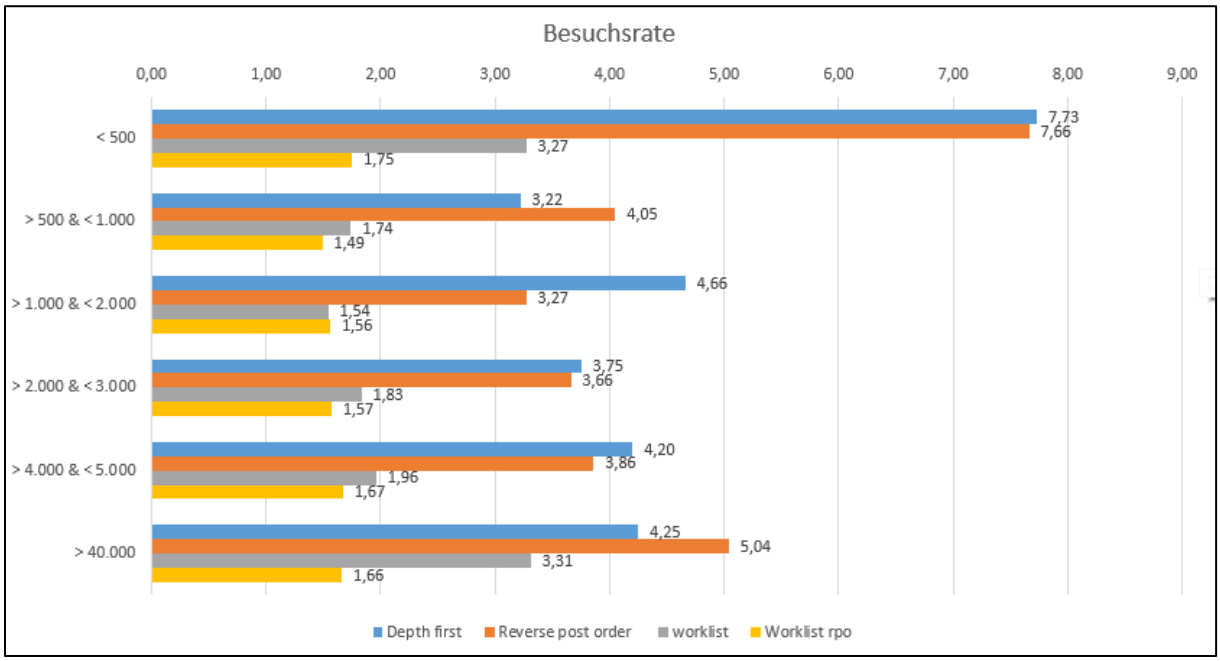

Abbildung 5.3.3.7: Besuchsrate alle Gruppen

Folgende Tabelle zeigt die Sieger aus der jeweiligen Gruppe:

Anzahl BBs	Laufzeit	Speicherverbrauch	Besuchsrate	BBs
< 500	worklist/ worklist rpo	alle	worklist rpo	worklist rpo
≥ 500 & < 1.000	worklist/ worklist rpo	alle	worklist rpo	worklist rpo
≥ 1.000 & < 2.000	DFO	alle	worklist	worklist
≥ 2.000 & < 3.000	DFO/ worklist rpo	alle	worklist rpo	worklist rpo
≥ 4.000 & < 5.000	DFO	alle	worklist rpo	worklist rpo
≥ 40.000	worklist rpo	worklist / worklist rpo	worklist rpo	worklist rpo

Tabelle 5.3.3.1: Sieger der jeweiligen Gruppe

Ein Blick auf die Kategorie Laufzeit zeigt, dass die Ansätze DFO und Worklist rpo am schnellsten sind. Jedoch sind die Unterschiede bei der Laufzeit der verschiedenen Ansätze durchgehend sehr gering. Was den Speicherverbrauch betrifft so unterscheiden sich die Ansätze nur für eine der sechs Kategorien.
Sowohl bei der Besuchsrate als auch bei der Anzahl der besuchten Grundblöcke schneidet der Worklist rpo-Ansatz in fünf von sechs Kategorien am besten ab. Die Worklist-Ansätze besuchen die wenigsten Grundblöcke, wobei der Worklist rpo-Ansatz am aller wenigsten Grundblöcke besucht.

Um nun zu einer Entscheidung zu kommen welcher Ansatz insgesamt am besten ist, brauchen wir eine Wertung. Vergeben wir nun für jeden Kategorie Sieg einen Punkt kommen wir zu folgender Wertung:

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	1	1	2	4
≥ 500 & < 1.000	1	1	2	4
≥ 1.000 & < 2.000	2	1	3	1
≥ 2.000 & < 3.000	2	1	1	4
≥ 4.000 & < 5.000	2	1	1	3
≥ 40.000	0	0	1	4

Tabelle 5.3.3.2: Wertung nach Gruppe

Die beste Wertung erzielt mit deutlichem Abstand der Ansatz Worklist rpo. An zweiter Stelle kommt der Worklist-Ansatz, welcher in der Wertung nur knapp vor dem DFO-Ansatz liegt, welcher den dritten Platz belegt.

5.3.4. Intraprozedural – Dominanz

Dieser Unterabschnitt befasst sich mit dem Fall der interprozeduralen Analyse für Dominanz.

Die Diagramme veranschaulichen, welcher Ansatz für welche Gruppe am besten abschneidet.

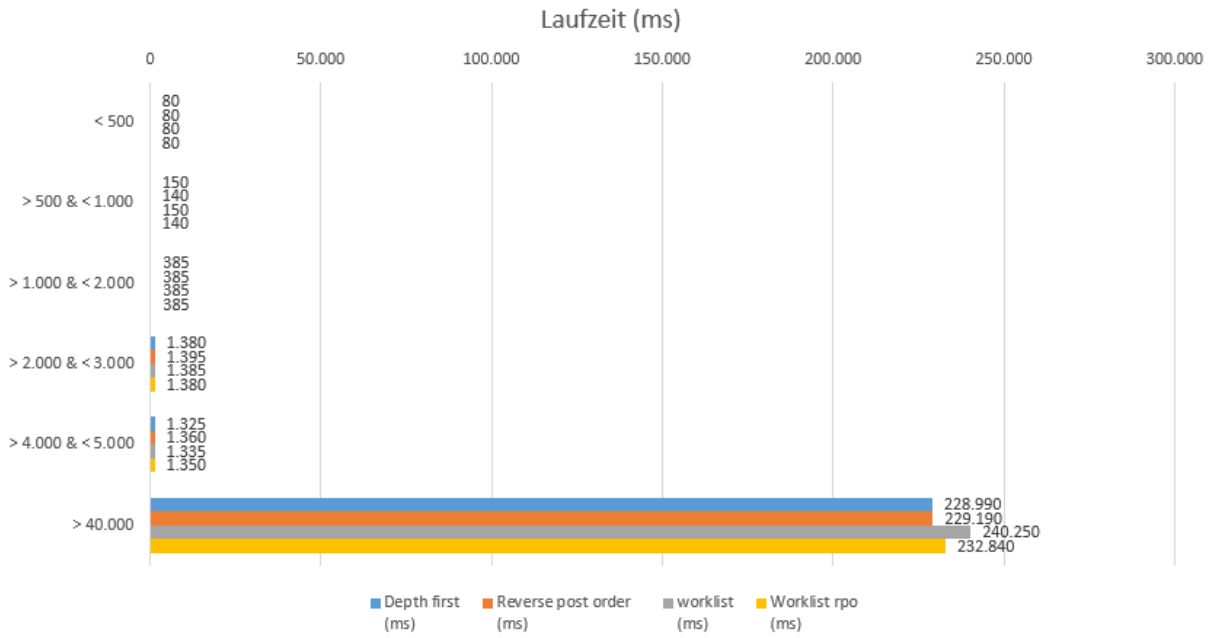

Abbildung 5.3.4.1: Laufzeit alle Gruppen

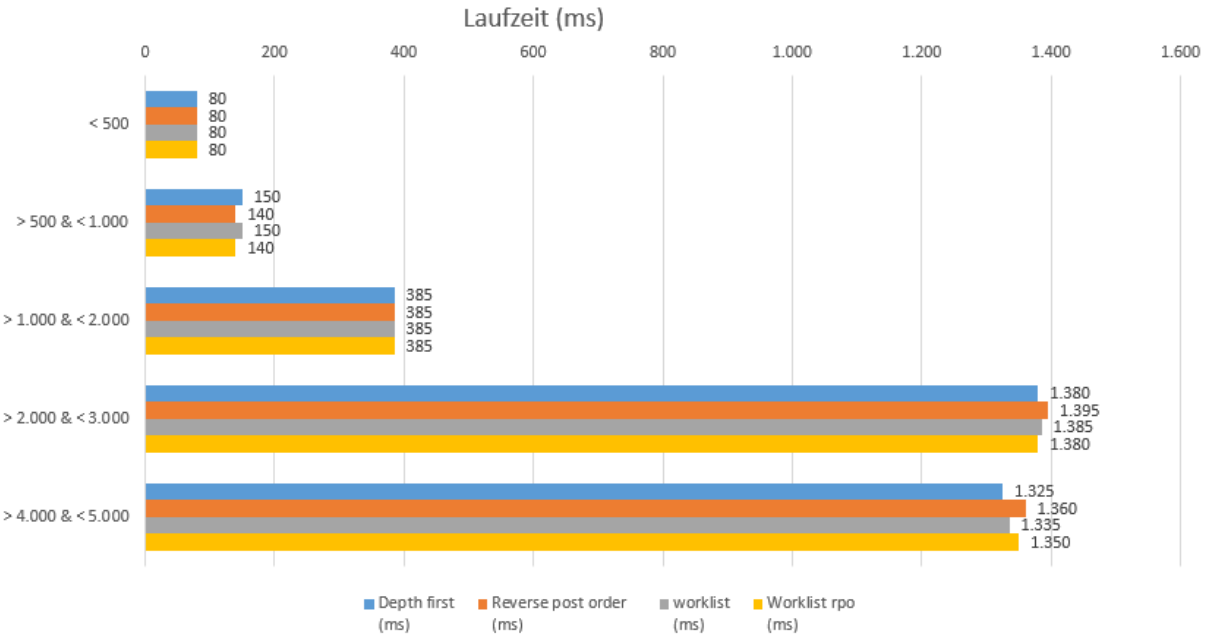

Abbildung 5.3.4.2: Laufzeit ohne größte Gruppe

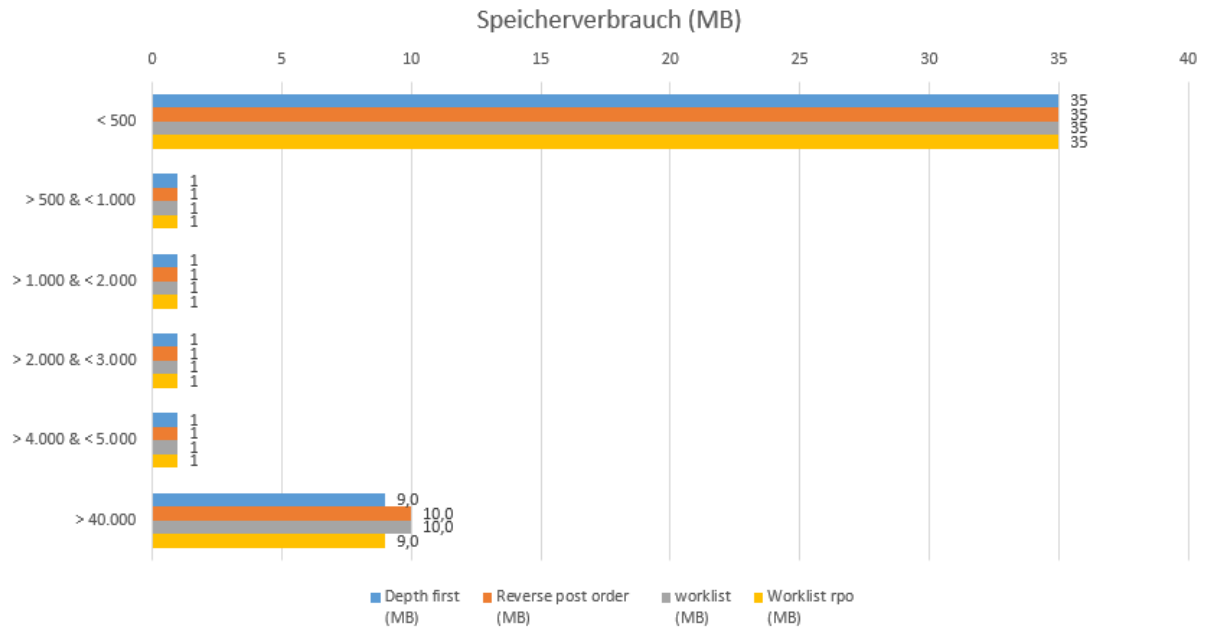

Abbildung 5.3.4.3: Speicherverbrauch alle Gruppen

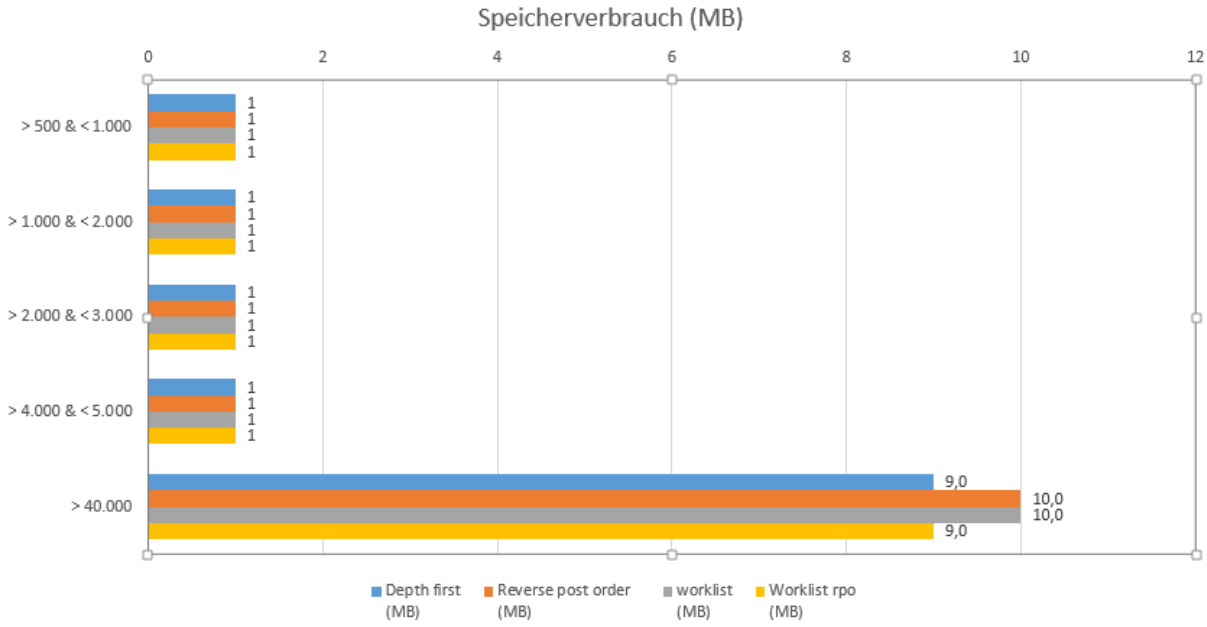

Abbildung 5.3.4.4: Speicherverbrauch ohne größte Gruppe

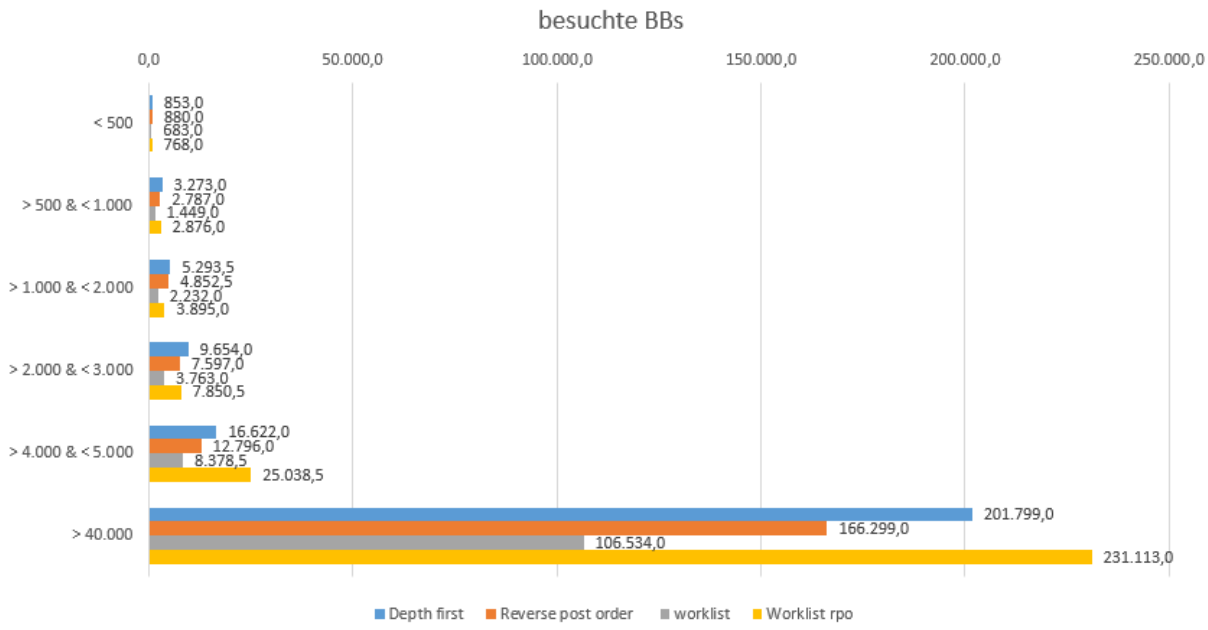

Abbildung 5.3.4.5: Anzahl besuchte Grundblöcke alle Gruppen

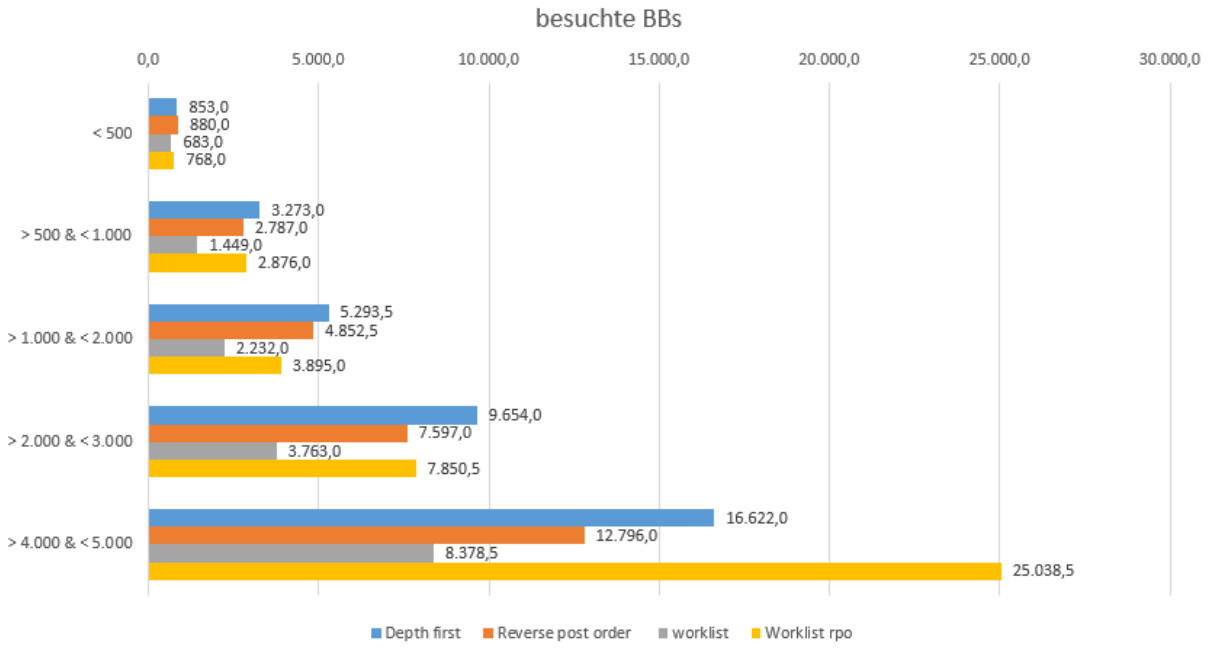

Abbildung 5.3.4.6: Anzahl besuchte Grundblöcke ohne größte Gruppe

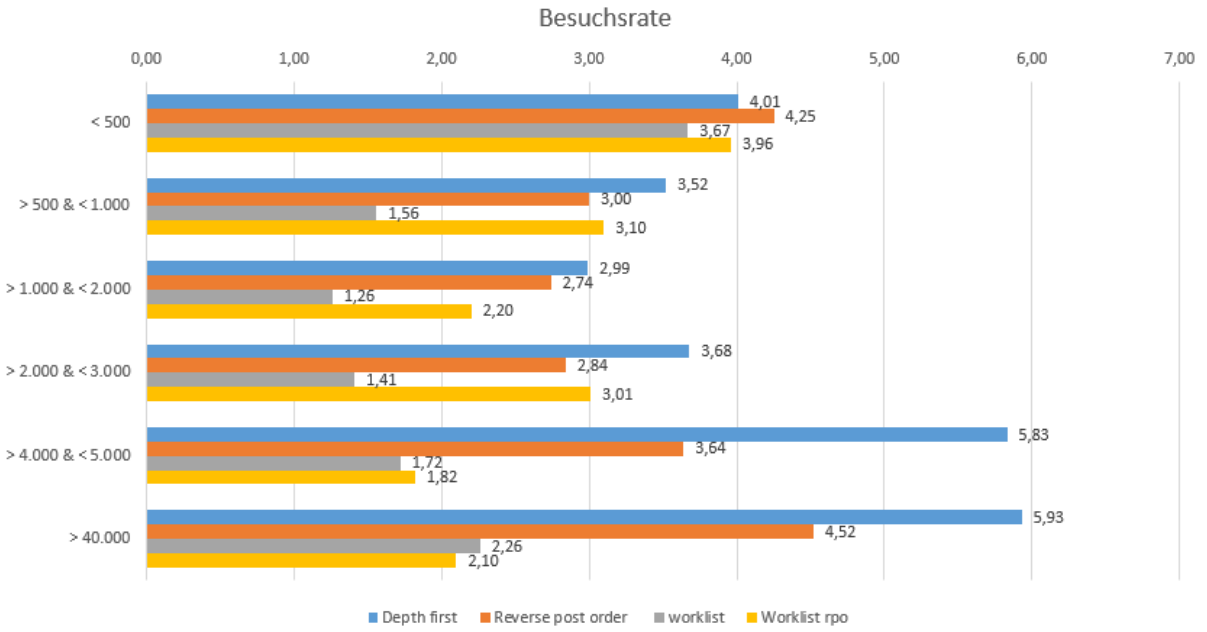

Abbildung 5.3.4.7: Besuchsrate alle Gruppen

Folgende Tabelle zeigt die Sieger aus der jeweiligen Gruppe:

Anzahl BBs	Laufzeit	Speicherver-brauch	Besuchsrate	BBs
< 500	alle	alle	worklist	worklist
≥ 500 & < 1.000	RPO/worklist rpo	alle	worklist	worklist
≥ 1.000 & < 2.000	alle	alle	worklist	worklist
≥ 2.000 & < 3.000	DFO/worklist rpo	alle	worklist	worklist
≥ 4.000 & < 5.000	DFO	alle	worklist	worklist
≥ 40.000	DFO	DFO/worklist rpo	worklist rpo	worklist

Tabelle 5.3.4.1: Sieger der jeweiligen Gruppe

Den Sieg in der Kategorie Laufzeit teilen sich die Ansätze DFO und Worklist rpo. Was den Speicherverbrauch betrifft, so unterscheiden sich die Ansätze nur für eine der sechs Kategorien. Bei der Besuchsrate schneidet der Worklist-Ansatz in fünf von sechs Kategorien am besten ab. Bei der Anzahl der besuchten Grundblöcke schneidet der Worklist-Ansatz sogar durchgehend am besten ab. Die Worklist-Ansätze besuchen die wenigsten Grundblöcke, wobei der Worklist-Ansatz, am aller wenigsten Grundblöcke besucht.

Um nun zu einer Entscheidung zu kommen welcher Ansatz insgesamt am besten ist, brauchen wir eine Wertung. Vergeben wir nun für jeden Kategorie Sieg einen Punkt kommen wir zu folgender Wertung:

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	2	2	4	2
≥ 500 & < 1.000	1	2	3	2
≥ 1.000 & < 2.000	2	1	4	2
≥ 2.000 & < 3.000	2	1	3	2
≥ 4.000 & < 5.000	2	1	3	1
≥ 40.000	2	0	1	2

Tabelle 5.3.4.2: Wertung nach Gruppe

In der Gesamtwertung liegt hier der Worklist-Ansatz deutlich an der Spitze. DFO und Worklist rpo teilen sich den zweiten Platz. RPO belegt hier den letzten Platz.

Sieht man sich die Resultate für Postdominanz und Dominanz an so fällt auf, so fällt folgendes auf.

Für Postdominanz schneidet der Ansatz Worklist rpo in der Gesamtwertung deutlich besser ab als die anderen Ansätze. Deutlich hervortun konnte sich der Ansatz Worklist rpo dabei jedoch nur bei der Besuchsrate und der Anzahl der besuchten Grundblöcke.
In Sachen Laufzeit und Speicherverbrauch liegen alle Ansätze so ziemlich gleich auf. Dies bestätigt auch die Erkenntnis aus [2], dass die Dominanz-Analysen laufzeitmäßig nicht von einer geringen Besuchszahl profitieren.

Für Dominanz erzielt der Worklist-Ansatz in der Gesamtwertung das bessere Ergebnis. Trotz deutlicher Führung in der Gesamtwertung kann sich auch der Worklist-Ansatz nur bei der Besuchsrate und der Anzahl der besuchten Grundblöcke positiv hervortun. In Sachen Laufzeit und Speicherverbrauch liegen auch hier alle Ansätze so ziemlich gleich auf. Damit bestätig sich auch hier die Erkenntnis aus [2], dass die Dominanz-Analysen laufzeitmäßig nicht von einer geringen Besuchszahl profitieren.

5.3.5. Interprozedural – Gültige Definitionen

Dieser Unterabschnitt befasst sich mit dem Fall der interprozeduralen Analyse für Gültige Definitionen.

Die Diagramme veranschaulichen, welcher Ansatz für welche Gruppe am besten abschneidet.

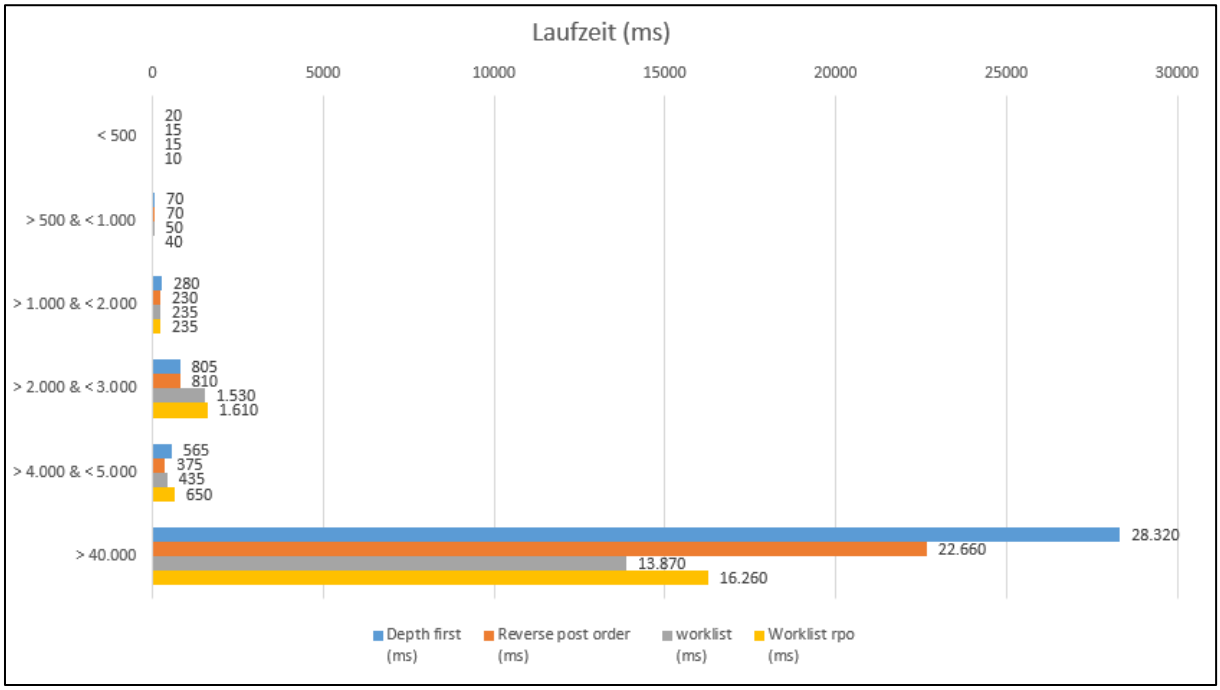

Abbildung 5.3.5.1: Laufzeit für alle Gruppen

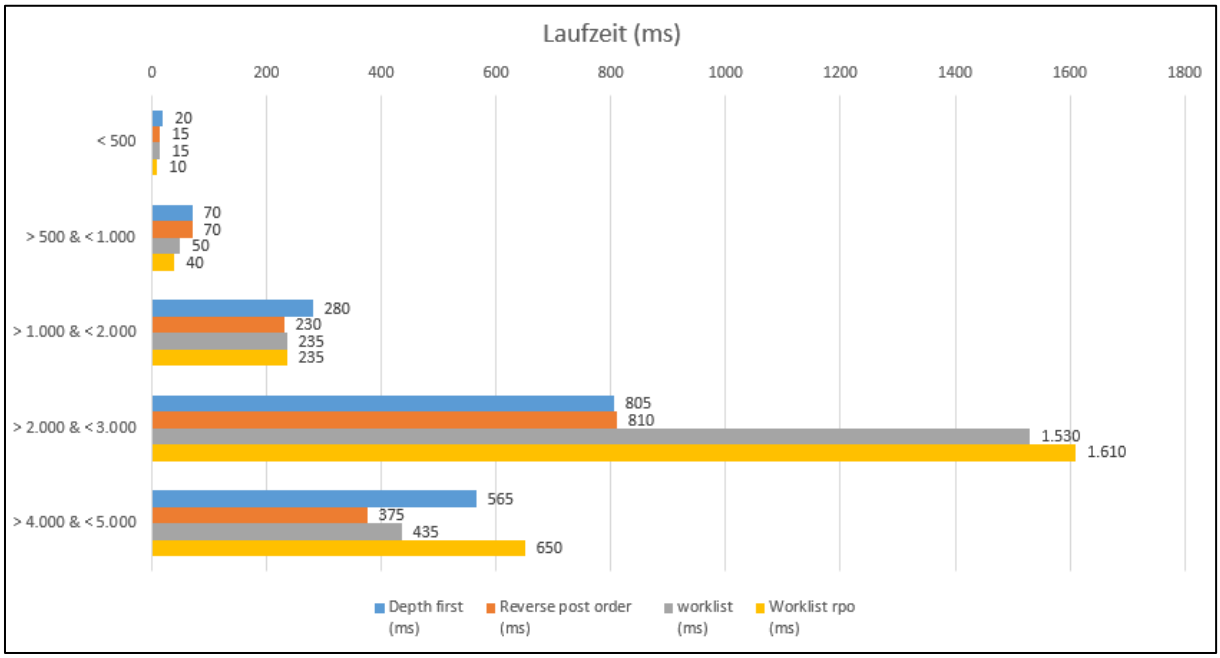

Abbildung 5.3.5.2: Laufzeit ohne größte Gruppe

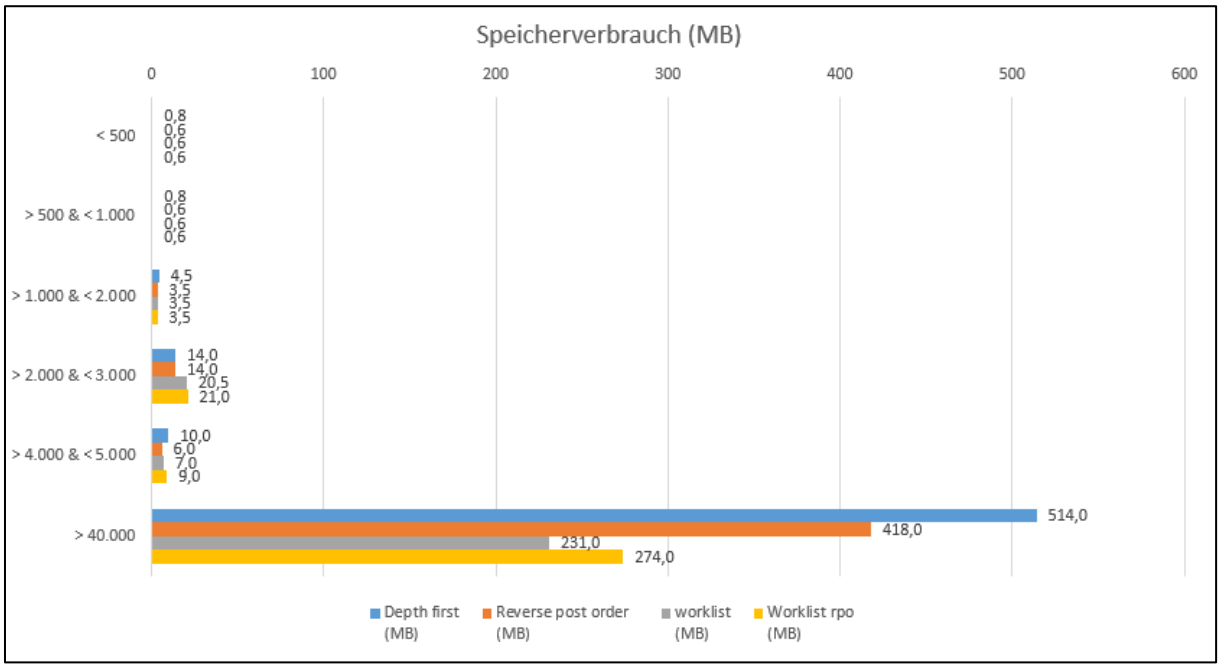

Abbildung 5.3.5.3: Speicherverbrauch alle Gruppen

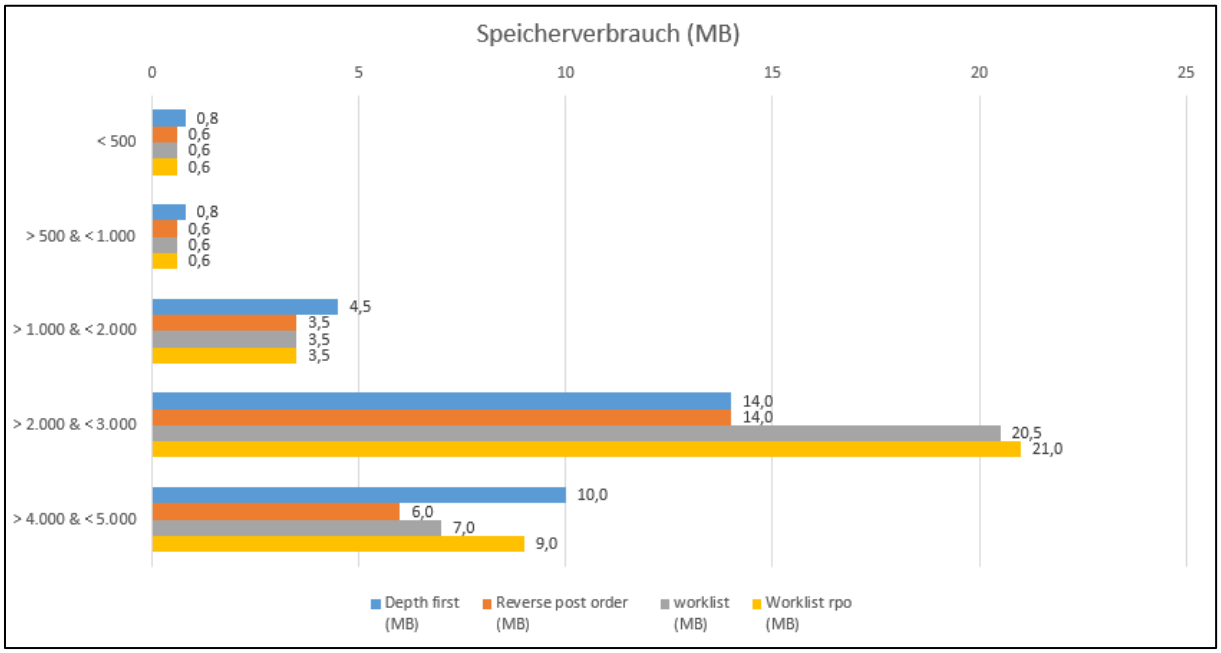

Abbildung 5.3.5.4: Speicherverbrauch ohne größte Gruppe

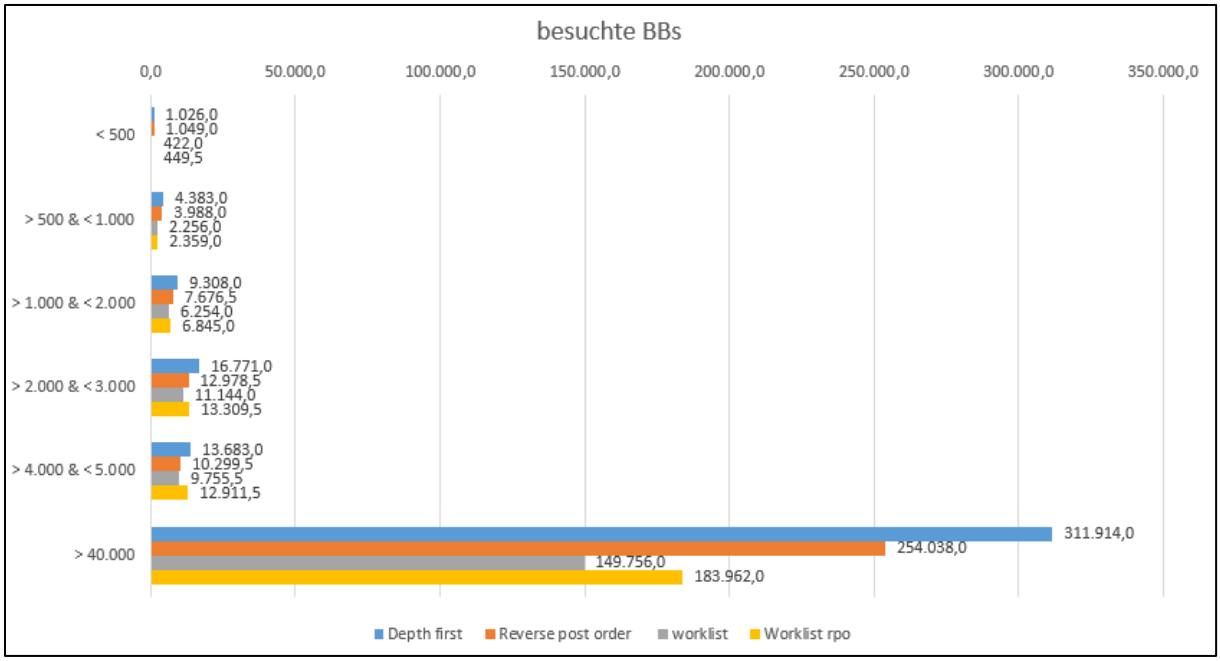

Abbildung 5.3.5.5: Anzahl besuchte Grundblöcke alle Gruppen

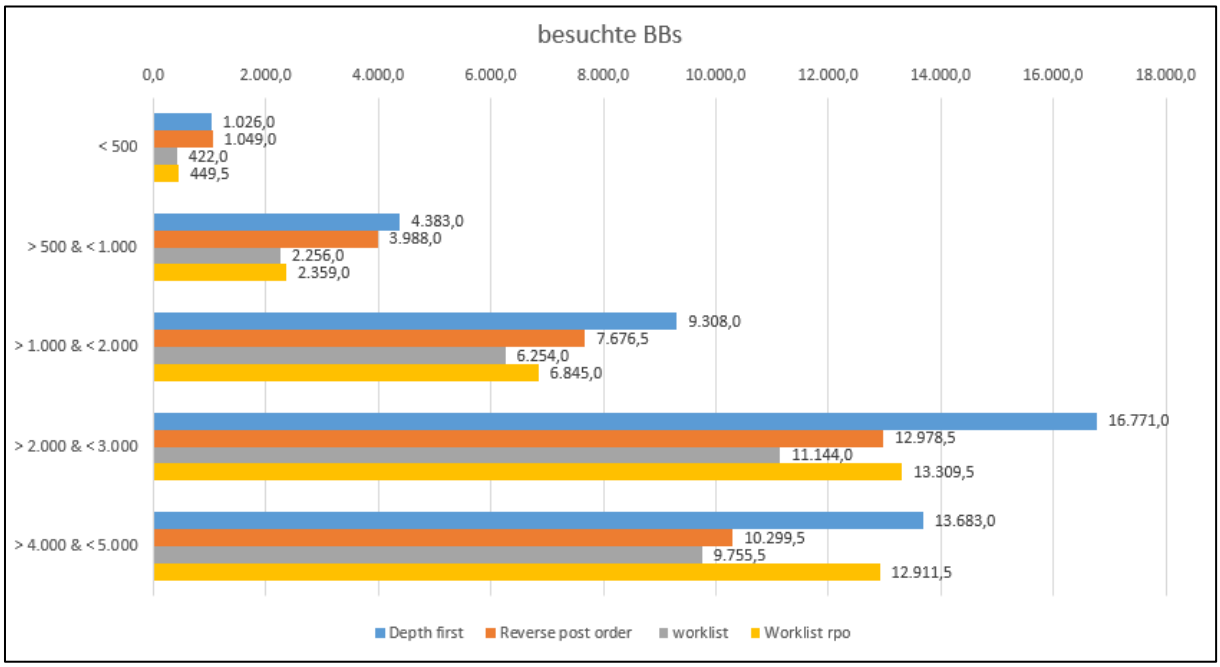

Abbildung 5.3.5.6: Anzahl besuchte Grundblöcke ohne größte Gruppe

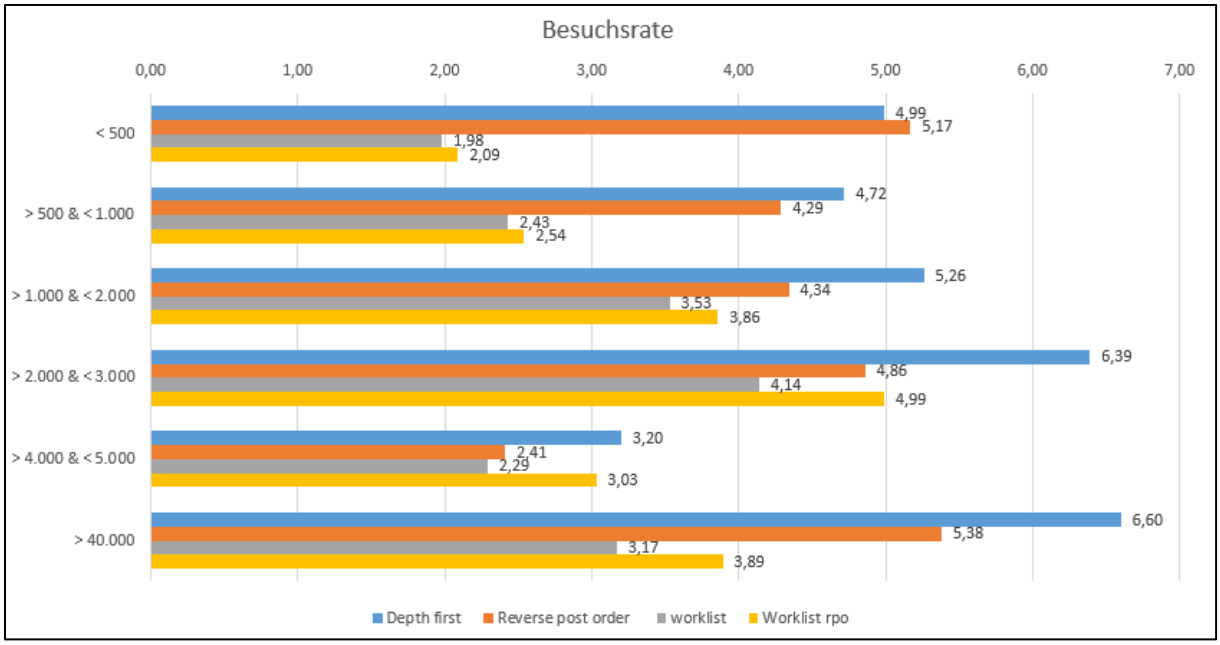

Abbildung 5.3.5.7: Besuchsrate alle Gruppen

Folgende Tabelle zeigt die Sieger aus der jeweiligen Gruppe:

Anzahl BBs	Laufzeit	Speicherverbrauch	Besuchsrate	BBs
< 500	worklist rpo	RPO/worklist/ worklist rpo	worklist	worklist
≥ 500 & < 1.000	worklist rpo	RPO/worklist/ worklist rpo	worklist	worklist
≥ 1.000 & < 2.000	RPO	RPO/worklist/ worklist rpo	worklist	worklist
≥ 2.000 & < 3.000	DFO	RPO/DFO	worklist	worklist
≥ 4.000 & < 5.000	RPO	RPO	worklist	worklist
≥ 40.000	worklist	worklist	worklist	worklist

Tabelle 5.3.5.1: Sieger der jeweiligen Gruppe

Wie man sieht wechseln sich in der Kategorie Laufzeit die Ansätze Worklist und RPO als Sieger ab. Der Worklist-Ansatz schneidet sowohl für kleine als auch sehr große Graphen am besten ab. Im Mittelfeld dagegen macht sich der RPO-Ansatz besser.
In der Kategorie Speicherverbrauch schneidet der RPO-Ansatz für alle Gruppen am besten, bzw. gleich gut wie die anderen Ansätze ab.
Sowohl bei der Besuchsrate, als auch bei der Anzahl der besuchten Grundblöcke ist der Worklist-Ansatz in allen Fällen am besten.
Um nun zu einer Entscheidung zu kommen welcher Ansatz insgesamt am besten ist, brauchen wir eine Wertung. Vergeben wir nun für jeden Kategorie Sieg einen Punkt kommen wir zu folgender Wertung:

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	0	1	3	2
≥ 500 & < 1.000	0	1	3	2
≥ 1.000 & < 2.000	0	2	3	1
≥ 2.000 & < 3.000	2	1	3	1
≥ 4.000 & < 5.000	0	2	2	0
≥ 40.000	0	0	4	0

Tabelle 5.3.5.2: Wertung nach Gruppe

Wie man sieht schneidet der Worklist-Ansatz in den meisten Fällen am besten ab. An zweiter Stelle kommt der RPO-Ansatz. Die anderen beiden Ansätze konnten sich nicht positiv hervortun.

5.3.6. Interprozedural – Lebendige Variablen

Dieser Unterabschnitt befasst sich mit dem Fall der interprozeduralen Analyse für Lebendige Variablen.

Die Diagramme veranschaulichen, welcher Ansatz für welche Gruppe am besten abschneidet.

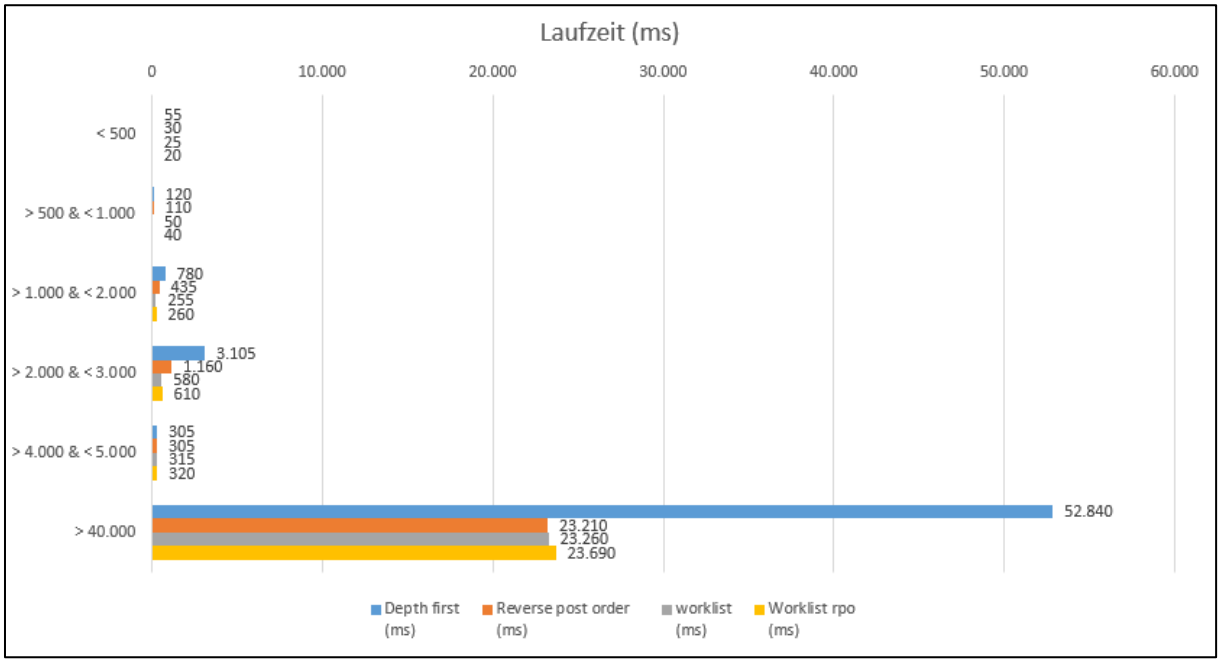

Abbildung 5.3.6.1: Laufzeit alle Gruppen

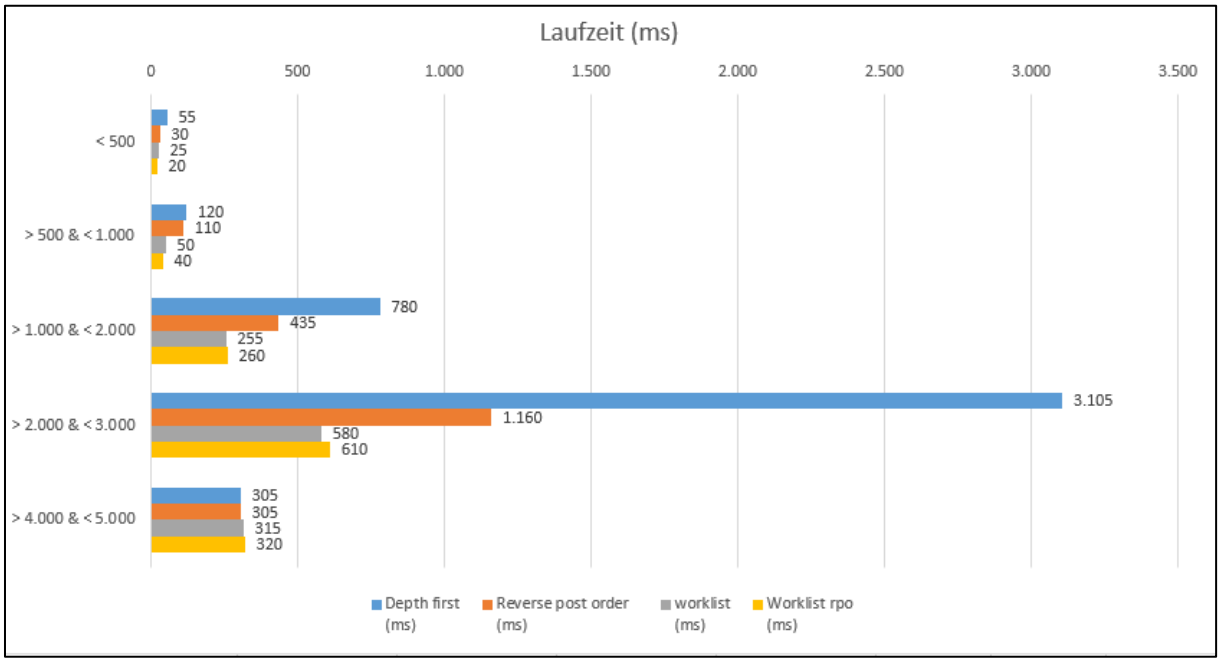

Abbildung 5.3.6.2: Laufzeit ohne größte Gruppe

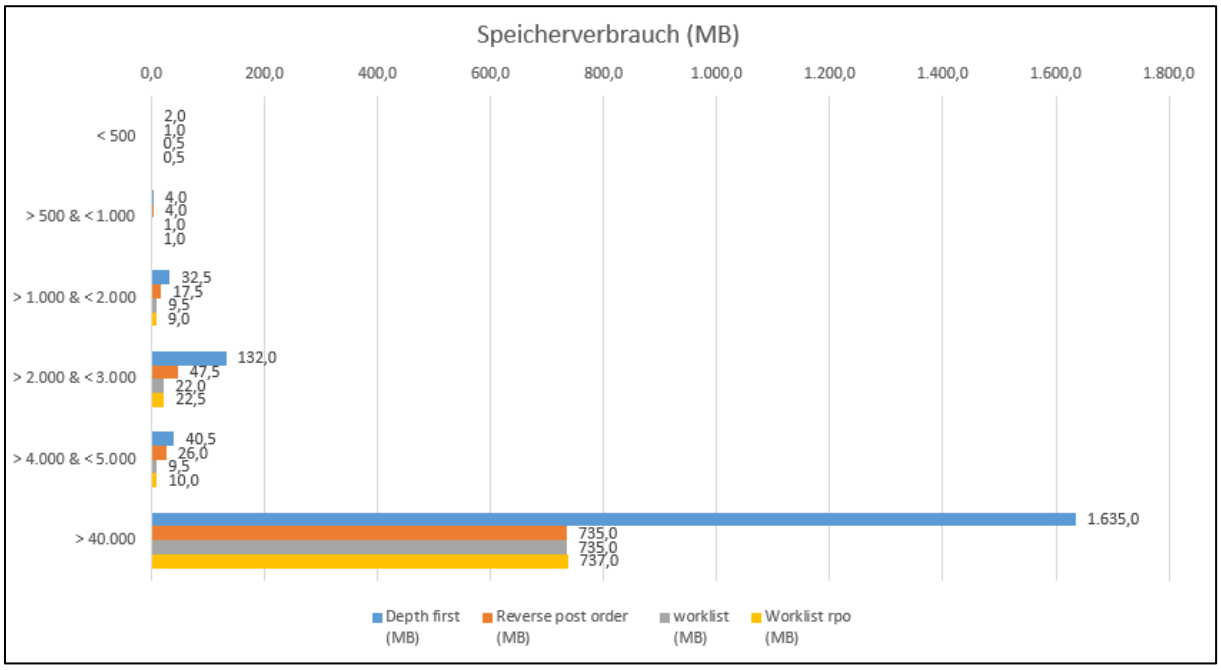

Abbildung 5.3.6.3: Speicherverbrauch alle Gruppen

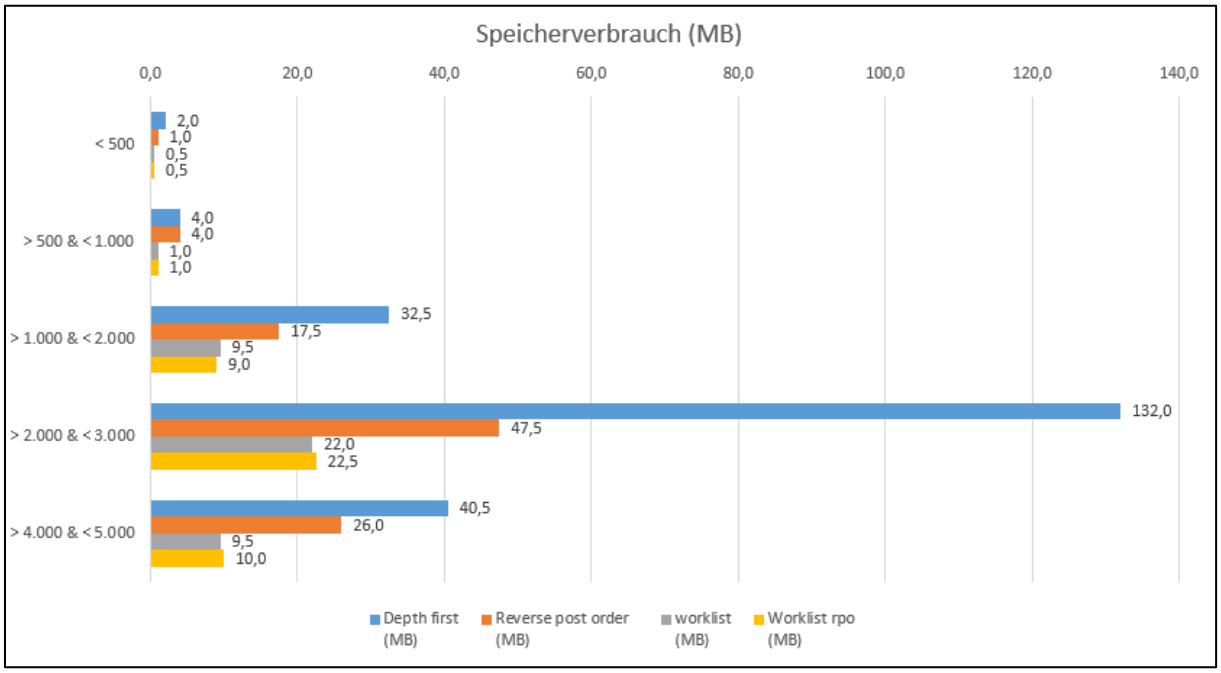

Abbildung 5.3.6.4: Speicherverbrauch ohne größte Gruppe

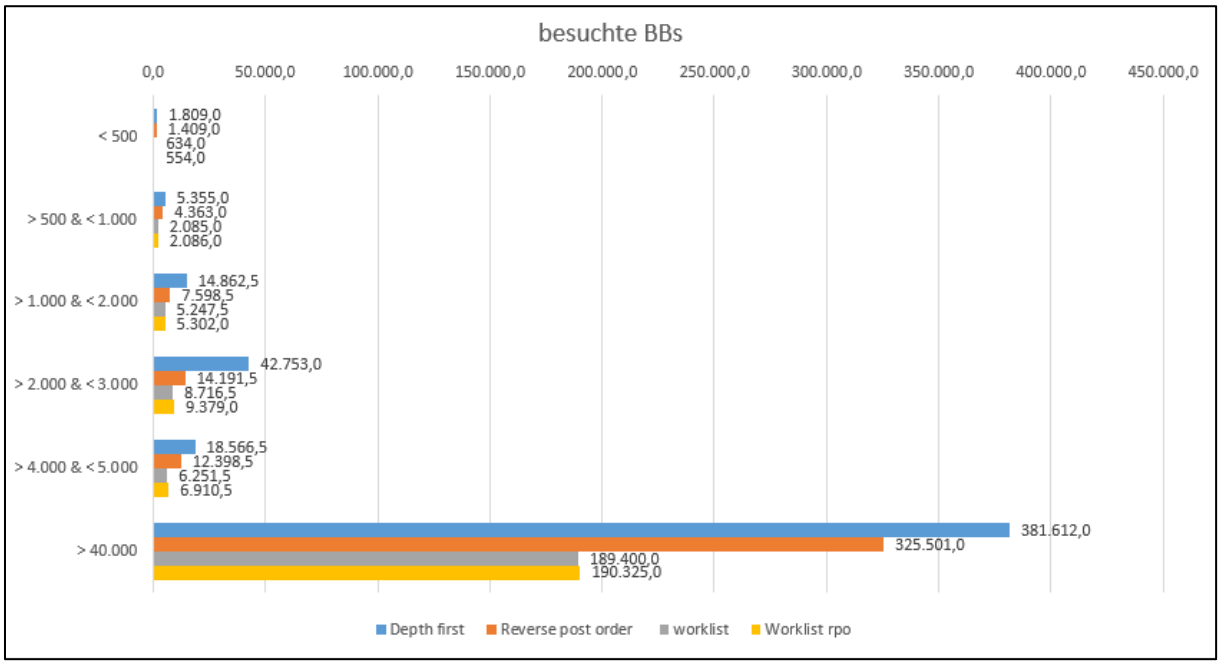

Abbildung 5.3.6.5: Anzahl besuchte Grundblöcke alle Gruppen

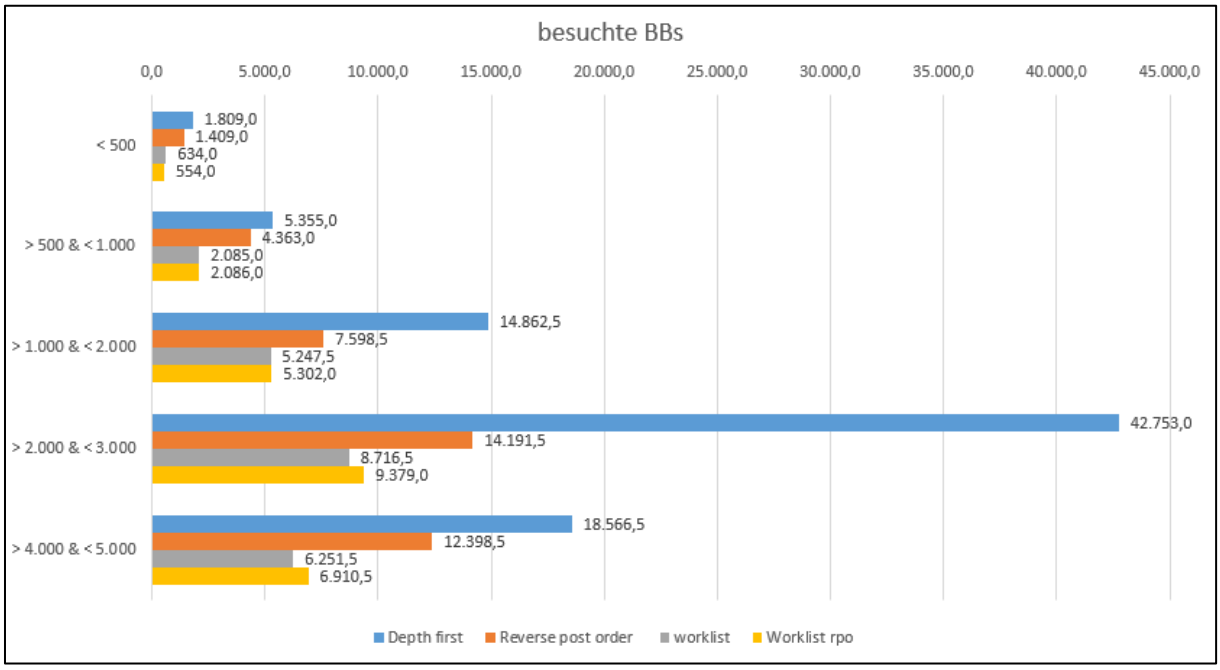

Abbildung 5.3.6.6: Anzahl besuchte Grundblöcke ohne größte Gruppe

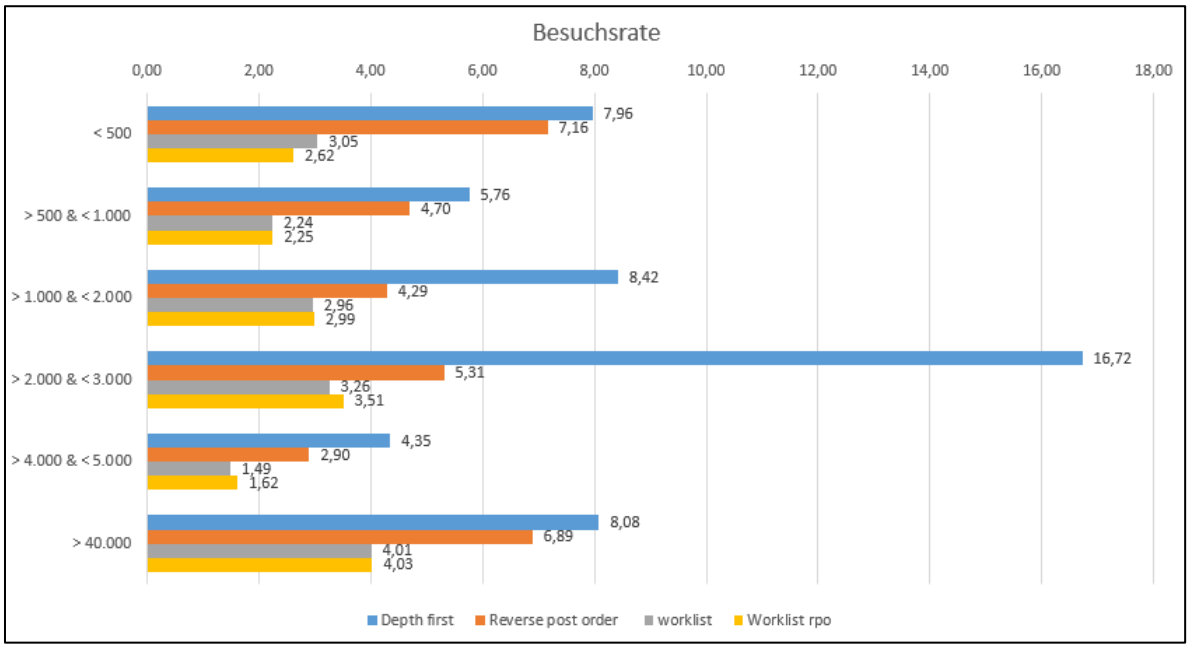

Abbildung 5.3.6.7: Besuchsrate alle Gruppen

Folgende Tabelle zeigt die Sieger aus der jeweiligen Gruppe:

Anzahl BBs	Laufzeit	Speicherverbrauch	Besuchsrate	BBs
< 500	worklist rpo	worklist/worklist rpo	worklist rpo	worklist rpo
≥ 500 & < 1.000	worklist rpo	worklist/worklist rpo	worklist	worklist
≥ 1.000 & < 2.000	worklist	worklist rpo	worklist	worklist
≥ 2.000 & < 3.000	worklist	worklist	worklist	worklist
≥ 4.000 & < 5.000	RPO/DFO	worklist	worklist	worklist
≥ 40.000	RPO	worklist / RPO	worklist	worklist

Tabelle 5.3.6.1: Sieger der jeweiligen Gruppe

Wie man sieht, wechseln sich in der Kategorie Laufzeit mehrere Ansätze als Sieger ab. Der Worklist rpo-Ansatz schneidet für die kleinsten der gemessenen Graphen am besten ab. Im Mittelfeld (1000 bis 3000 Grundblöcke) macht sich der Worklist-Ansatz besser, während ab 4000 Grundblöcken die RPO- und DFO-Ansätze besser abschneiden.
In der Kategorie Speicherverbrauch schneiden der Worklist-Ansatz und der Worklist rpo-Ansatz für fast alle Gruppen am besten ab. Bei der Besuchsrate und der Anzahl der besuchten Grundblöcke ist der Worklist-Ansatz in fünf von sechs Fällen am besten.

Um nun zu einer Entscheidung zu kommen, welcher Ansatz insgesamt am besten ist, brauchen wir eine Wertung. Vergeben wir nun für jeden Kategorie Sieg einen Punkt kommen wir zu folgender Wertung:

Anzahl BBs	Depth-first	Reverse Postorder	worklist	worklist rpo
< 500	0	0	1	4
≥ 500 & < 1.000	0	0	3	2
≥ 1.000 & < 2.000	0	0	3	1
≥ 2.000 & < 3.000	0	0	4	0
≥ 4.000 & < 5.000	1	1	3	0
≥ 40.000	0	2	3	0

Tabelle 5.3.6.2: Wertung nach Gruppe

Wie man sieht, schneidet der Worklist-Ansatz in den meisten Fällen am besten ab. An zweiter Stelle kommt der Worklist rpo-Ansatz. Den dritten Platz und den letzten Platz belegen mit Abstand der RPO- und der DFO-Ansatz. Vergleichung wir nun die Wertungen der Analyse

Gültige Definitionen mit der Wertung der Analyse Lebendige Variablen für den interprozeduralen Fall. Hier fällt auf, dass die Wertungen für beide Analysen ähnlich ausfallen.

Worklist geht in beiden Fällen klar als Sieger hervor, während DFO in beiden Fällen mit nur wenigen Punkten auf dem letzten Platz landet.

5.4. Fazit

Nachdem wir nun zu einer Wertung gekommen sind, können wir jetzt die Eingangs gestellten Fragen beantworten.

Beginnen wir mit der Frage, ob es einen Ansatz gibt, der für alle Datenflussprobleme gleichermaßen geeignet ist.
In den Gesamtwertungen für die untersuchten Datenflussprobleme schnitten die beiden Worklist-Ansätze am besten ab. Dieses Ergebnis deckt sich für die Datenflussprobleme Lebendige Variablen und Dominanz mit den Ergebnissen aus [2]. Das könnte als Hinweis darauf gedeutet werden, dass der beste Ansatz für alle Datenflussprobleme unter den Worklist-Ansätzen zu finden ist.
Dagegen spricht jedoch, dass sowohl die Größe der Graphen, als auch die Art des Datenflussproblems einen Unterschied ausmacht, welcher der Worklist-Ansätze besser abschneidet. Auch hier deckt sich das Resultat für die Datenflussprobleme Lebendige Variablen und Dominanz mit den Ergebnissen aus [2].
Die Gesamtwertungen sprechen damit dagegen, dass es einen Ansatz gibt, der für alle Datenflussprobleme gleichermaßen geeignet ist.

Befassen wir uns deshalb mit der zweiten Frage, welcher Ansatz für welche Datenflussprobleme am besten geeignet ist.
In den Gesamtwertungen für das Datenflussproblem Lebendige Variablen schneidet für die kleinsten gemessenen Graphen sowohl im intra-, als auch im interprozeduralen Fall der Worklist rpo-Ansatz laufzeitmäßig am besten ab. Für die größeren gemessenen Graphen schneidet der Worklist-Ansatz mit Stack besser ab.
Damit entspricht das Resultat auch hier weitgehend den Ergebnissen aus [2].

Der Vergleich der Gesamtergebnisse der Analysen Gültige Definitionen und Lebendige Variablen deutet darauf hin, dass die Ansätze Worklist und Worklist rpo im intraprozeduralen Fall für die meisten Analysen am besten geeignet sind. Bei Graphen mittlerer Größe sind jedoch die Ansätze RPO und DFO geringfügig besser im Laufzeitverhalten, RPO schneidet zusätzlich auch beim Speicherverbrauch besser ab.
Vergleicht man hier den interprozeduralen Fall, kommt man zu ähnlichen Schlüssen. Auch hier erscheinen die Ansätze Worklist und Worklist rpo für die meisten Analysen am besten geeignet. Die Ansätze RPO und DFO schneiden diesmal für die größeren gemessenen Graphen im Laufzeitverhalten besser ab.

Nach den Gesamtwertungen für das Datenflussproblem Dominanz schneiden für die kleinsten gemessenen Graphen alle Ansätze annähernd gleich gut ab. Auch für die größeren gemessenen Graphen sind die Ansätze laufzeitmäßig immer noch annähernd gleich schnell.
Die Ergebnisse für das Datenflussproblem Postdominanz sehen erwartungsgemäß ähnlich aus. Auch hier schneiden alle Ansätze sowohl bei den kleinsten, als auch bei größeren gemessenen Graphen laufzeitmäßig annähernd gleich gut ab.

Das Resultat aus [2], dass das Datenflussproblem Lebendige Variablen im Gegensatz zum Datenflussproblem Dominanz laufzeitmäßig von wenigen besuchten Grundblöcken profitiert, deckt sich auch hier mit den Gesamtwertungen.

6. Zusammenfassung und Ausblick

In diesem Kapitel wird das Ergebnis der Diplomarbeit aus der persönlichen Sicht des Autors geschildert. Auch wird ein Ausblick auf mögliche Verbesserungen und Erweiterungen gegeben.

6.1. Bewertung des Projekts

In diesem Abschnitt gehe ich auf die Probleme ein, die währen der einzelnen Phasen der Arbeit auftraten. Auch gehe ich auf die hieraus resultierenden Änderungen für den weiteren Verlauf der Arbeit ein.
In der Anfangsphase, bestehend aus Einarbeitung, Bestandsaufnahme und Projektplanung traten keine größeren Probleme auf. Erste Probleme ergaben sich erst, als ich mit der Implementierung auf einem Stand war, dass die Implementierung alle Tests erfolgreich durchläuft.

Zu diesem Zeitpunkt lag ich noch gut innerhalb der für die Implementierung eingeplanten Zeit. Dann stellte sich heraus, dass mit der bisherigen Implementierung keine interprozeduralen Analysen möglich waren. Die Implementierung musste daher nochmals komplett überarbeitet werden, wodurch sich die Implementierungsphase deutlich verlängerte und sich die nachfolgenden Phasen entsprechend verschoben. Damit war der eingeplante Zeitpuffer dann aufgebraucht.

Die Phase des funktionalen Tests konnte deshalb erst deutlich verspätet begonnen werden. In dieser Phase waren dann nur kleinere Anpassungen erforderlich, so dass sich der Projektverlauf nur geringfügig weiter verzögerte. Weitere Probleme ergaben sich bei der Erzeugung der IML-Dateien für die Testprogramme. Die Tests mit realen Programmen konnten deshalb erst erheblich später durchgeführt werden. Bis die Implementierung dann auch mit realen Programmen rund lief, verging noch einmal einige Zeit.

In der Phase der Messung kam es zu Beginn noch an etlichen Stellen zu Speicherausfällen. Diese ursprüngliche Ineffizienz der Implementierung war maßgeblich auf zu häufige If-Abfragen zurückzuführen. Nachdem ich die Implementierung dahingehend nachgebessert hatte, traten keine Speicherausfälle mehr auf und ich konnte die Messungen vollständig und ohne Ausfälle durchführen.

Leider stellte sich viel zu spät heraus, dass die Implementierung sowohl mit Pointern als auch mit Selektionen nicht richtig umgehen kann, da diese in der IML anders strukturiert sind und daher gesondert behandelt werden müssten. Das Problem bei den Pointern ist, dass sowohl Pointer-Ziele als auch Pointer-Deallokationen nicht berücksichtigt werden. Lediglich die Pointer-Variablen selbst werden erfasst, jedoch wie normale Variablen behandelt. Selektionen werden von der Implementierung gar nicht erfasst.

Der fehlende Umgang mit Pointern und Selektionen verändert die Messergebnisse nur größenmäßig. Auf die prinzipielle Korrektheit der Messergebnisse hat dies aber keinen Einfluss. Die für den Umgang mit Pointern und Selektionen notwendigen Änderungen an der Implementierung wurden aus Zeitgründen nicht mehr vorgenommen. Der Grund dafür liegt darin, dass diese erheblichen Zeitaufwand in Form von Tests und Neumessungen erfordert hätten. Die Nachforschungen zu einem anderen Problem erschienen mir wichtiger.

Bei den Messergebnissen ist mir aufgefallen, dass der Worklist rpo-Ansatz stellenweise mehr Grundblöcke besucht als der Worklist-Ansatz. Die höheren Besuchszahlen des Worklist rpo-Ansatzes stellen ein mögliches Problem für die Korrektheit der Implementierung des Worklist-rpo-Ansatzes dar, da sie der Theorie widersprechen.
Es wurde daher eine Suche nach einem Fehler in der Implementierung erforderlich. Im Rahmen der Suche nach Fehlern konnten die wahrscheinlichsten Ursachen für den Fehler ausgeschlossen werden. Die Ergebnisse der Ursachenforschung für dieses Problem wurden in 5.2.4 dokumentiert. Die höheren Besuchszahlen des Worklist rpo-Ansatzes ließen sich damit jedoch nicht erklären.

6.2. Ergebnisse

Ergebnisse dieser Diplomarbeit sind:

- **Empirische Bewertung der Ansätze DFO, RPO, Worklist und Worklist rpo für die Datenflussprobleme Gültige Definitionen, Lebendige Variablen, Postdominanz und Dominanz.**
- Skeleton mit dem sich Datenflussanalysen so implementieren lassen, dass sich die Ansätze einfach austauschen lassen.
- Implementierung der Analysen Gültige Definitionen, Lebendige Variablen, Postdominanz und Dominanz für das Skeleton.
- Erheblicher Lern-Zugewinn des Autors mit Ada, IML und Datenflussanalysen.

6.3. Erweiterungen und Verbesserungen

Für das Skeleton und die implementierten Analysen sind viele Erweiterungen und Verbesserungen möglich.

1. **Erweiterungen, die keine Änderungen an der Architektur erfordern**
    a. Es könnten noch weitere Analysen mit dem Skeleton implementiert werden.
    b. Die bereits vorhandenen Analysen könnten effizienter implementiert werden.
    c. Eine Erweiterung der Implementierung der Analysen, so dass sie sowohl mit Pointern als auch mit Selektionen richtig umgehen können.
2. **Erweiterungen, die minimale Änderungen an der Architektur erfordern**
    a. Eine auf Splittung der Methoden im Package GDFA_Analysis nach der Richtung der Analyse, verspricht noch mal eine deutliche Effizienzsteigerung
3. **Erweiterungen, die größere Änderungen an der Architektur erfordern**
    a. Es könnte interessant sein, andere Umsetzungen für die interprozedurale Erweiterung auszuprobieren.

7. Literaturverzeichnis

[1] K. D. Cooper, T. J. Harvey und K. Kennedy, „A Simple, Fast Dominance Algorithm," Department of Computer Science, Rice University , Houston, Texas , USA, 2001.

[2] K. D. Cooper, T. J. Harvey und K. Kenndy, „Iterative data-flow analysis, revisited," Department of Computer Science, Rice University Houston, Texas, USA, 2002.

[3] A. Raza, G. Vogel und E. Plödereder, „Bauhaus – a Tool Suite for Program Analysis and Reverse Engineering," in *Ada Europe Ausgabe 4006*, Universität Stuttgart, Institut für Softwaretechnologie, Universitätsstraße 38 70569 Stuttgart: Lecture Notes in Computer Science, 2006, pp. 71-82.

[4] M. Würthner, „Entwurf und Implementierung einer Interndarstellung für die Analyse von Ada-Programmen.," Studienarbeit Nr. 1567, Universität Stuttgart, 1996.

[5] J. Rohrbach, „Erweiterung und Generierung einer Zwischendarstellung für C-Programme," Studien Arbeit Nr. 1662, Universität Stuttgart, 1998.

[6] M. Knauß, „Erweiterung und Generierung der Zwischendarstellung IML für Java-Programme," Diplomarbeit Nr. 2006, Universität Stuttgart, 2002.

[7] S. S. T. Karaca, „Erweiterung und Generierung der Zwischendarstellung IML für C++ Programme," Diplomarbeit Nr. 2048, Universität Stuttgart, 2003.

[8] S. Keul, „Generierung der Zwischendarstellung IML für Ada 95 Programme," Diplomarbeit Nr. 2323, Universität Stuttgart, 2005.

[9] A. V. Aho, M. S. Lam, R. Sethi und J. D. Ullman, Compiler Prinzipien, Techniken und Werkzeuge, Deutschland: Person Studium, 2010.

[10] U. P. Khedker, A. Sanyal und B. Karkare, Data Flow Analysis Theory and Practice, United States of America: CRC Press, 2009.

[11] F. Nielson, H. R. Nielson und C. Hankin, Principles of Program Analysis, Berlin: Springer, 2005.

[12] S. Staiger-Stöhr, „Kombinierte statische Ermittling von Zeigerzielen, Kontroll- und Datenfluss," Dissertation, Stuttgart , 2009.

[13] S. Z. G. a. C. L. Teck Bok Tok, „Efficient Flow-Sensitive Interprocedural Data-Flow Analysis in the Presence of Pointers," Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712, USA 2 Department of Computer Science, Tufts University, Medford, MA 02155, USA , 2006.

8. Anhang A - Messwerte

Anmerkungen:

- Depth-first wird in der Präfix Variante verwendet
- Depth-first und reverse Postorder arbeiten für die Vorwärtsanalyse auf dem Kontrollflussgraphen
- Depth-first und reverse Postorder arbeiten für die Rückwärtsanalyse auf dem inversen Kontrollflussgraphen
- Der Speicherverbrauch ist bereinigt, das bedeutet es wurde alles herausgerechnet, was vor der Analyse schon im Speicher war (z.B. der eingelesen Graph)

Programm	Routines	BBs	calls
time.tt	25	283	140
Mkeyes.tt	4	166	47
Darkhttpd.tt	58	929	427
dc.tt	127	1.752	630
Concepts.tt	176	1.791	654
Joseki.tt	258	4.350	733
Gnuplot_x11	97	2.416	847
Bc.tt	149	2.848	1.037
Nano.tt	212	4.241	2.268
Gnugo.tt	2.989	47.242	11.961

Tabelle 8.0: Größe der analysierten Programme

8.1. Reaching Defs

8.1.1. Intra Procedural

- **Laufzeit**

Programm	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
Time.tt	10	10	10	10
Mkeyes.tt	40	30	10	10
Darhttpd.tt	40	50	20	40
dc.tt	80	60	50	60
Concepts.tt	140	120	180	180
Joseki.tt	160	130	90	160
Gnuplot_x11.tt	180	90	50	100
Bc.tt	540	520	1.850	1.970
Nano.tt	600	560	980	1.150
Gnugo.tt	442.320	442.910	238.650	242.380

Tabelle 8.1.1.1: Übersicht Laufzeit

- **Speicherverbrauch**

Programm	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
Time.tt	< 1	< 1	< 1	< 1
Mkeyes.tt	< 1	< 1	< 1	< 1
Darhttpd.tt	< 1	< 1	< 1	< 1
dc.tt	1	1	< 1	< 1
Concepts.tt	2	2	3	3
Joseki.tt	2	1	1	2
Gnuplot_x11.tt	2	1	< 1	1
Bc.tt	7	7	17	17
Nano.tt	11	10	18	21
Gnugo.tt	126	108	137	173

Tabelle 8.1.1.2: Übersicht Speicherverbrauch

- **Besuchte BBs**

Programm	Depth-first	Reverse Postorder	worklist	worklist rpo
Time.tt	658	724	387	407
Mkeyes.tt	1.108	1.185	403	410
Darhttpd.tt	3.439	2.954	1.644	1.782
dc.tt	6.379	4.386	2.367	2.991
Concepts.tt	5.911	5.439	4.447	4.733
Joseki.tt	13.913	10.721	5.526	10.411
Gnuplot_x11.tt	10.393	5.420	2.730	4.298
Bc.tt	11.642	10.024	14.977	16.053
Nano.tt	18.320	15.038	16.039	19.885
Gnugo.tt	203.576	176.321	112.355	145.894

Tabelle 8.1.1.3: Übersicht besuchte Grundblöcke

8.1.2. Inter Procedural

- **Laufzeit**

Programm	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
Time.tt	10	10	10	10
Mkeyes.tt	30	20	20	10
Darhttpd.tt	70	70	50	40
dc.tt	220	170	130	150
Concepts.tt	340	290	340	320
Joseki.tt	130	80	50	110
Gnuplot_x11.tt	440	290	230	360
Bc.tt	1.170	1.330	2.830	2.860
Nano.tt	1.000	670	820	1.190
Gnugo.tt	28.320	22.660	13.870	16.260

Tabelle 8.1.2.1: Übersicht Laufzeit

- **Speicherverbrauch**

Programm	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
Time.tt	< 1	< 1	< 1	< 1
Mkeyes.tt	< 1	< 1	< 1	< 1
Darhttpd.tt	< 1	1	< 1	< 1
dc.tt	3	2	1	2
Concepts.tt	6	5	6	5
Joseki.tt	2	1	< 1	1
Gnuplot_x11.tt	7	5	3	5
Bc.tt	21	23	38	37
Nano.tt	18	11	13	17
Gnugo.tt	514	418	231	274

Tabelle 8.1.2.2: Übersicht Speicherverbrauch

- **Besuchte BBs**

Programm	Depth-first	Reverse Postorder	worklist	worklist rpo
Time.tt	953	925	450	498
Mkeyes.tt	1.099	1.173	394	401
Darhttpd.tt	4.383	3.988	2.256	2.359
dc.tt	10.293	7.927	6.091	6.933
Concepts.tt	8.323	7.426	6.417	6.757
Joseki.tt	8.044	5.303	3.182	5.871
Gnuplot_x11.tt	15.859	9.795	7.244	10.097
Bc.tt	17.683	16.162	15.043	16.522
Nano.tt	19.322	15.296	16.329	19.952
Gnugo.tt	311.914	254.038	149.756	183.962

Tabelle 8.1.2.3: Übersicht besuchte Grundblöcke

8.2. Live Variables

8.2.1. Intra Procedural

- **Laufzeit**

Programm	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
Time.tt	30	20	10	10
Mkeyes.tt	150	220	60	50
Darhttpd.tt	40	40	50	40
dc.tt	90	90	90	90
Concepts.tt	60	60	60	60
Joseki.tt	390	400	400	410
Gnuplot_x11.tt	360	360	360	360
Bc.tt	200	200	220	210
Nano.tt	220	210	230	230
Gnugo.tt	241.120	234.120	226.570	225.610

Tabelle 8.2.1.1: Übersicht Laufzeit

- **Speicherverbrauch**

Programm	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
Time.tt	< 1	1	< 1	1
Mkeyes.tt	< 1	< 1	< 1	< 1
Darhttpd.tt	1	1	1	1
dc.tt	1	1	1	1
Concepts.tt	1	1	1	1
Joseki.tt	7	7	7	7
Gnuplot_x11.tt	4	3	4	4
Bc.tt	2	2	2	2
Nano.tt	4	4	4	4
Gnugo.tt	817	740	261	209

Tabelle 8.2.1.2: Übersicht Speicherverbrauch

- **Besuchte BBs**

Programm	Depth-first	Reverse Postorder	worklist	worklist rpo
Time.tt	1.722	1.012	453	427
Mkeyes.tt	1.265	1.763	656	542
Darhttpd.tt	4.554	3.543	1.514	1.484
dc.tt	14.990	4.674	2.250	2.331
Concepts.tt	7.132	6.044	3.301	3.248
Joseki.tt	21.669	13.298	6.800	7.700
Gnuplot_x11.tt	44.807	5.999	2.915	3.291
Bc.tt	22.078	12.487	5.479	5.271
Nano.tt	28.326	17.919	7.972	7.961
Gnugo.tt	280.218	213.720	106.534	99.148

Tabelle 8.2.1.3: Übersicht besuchte Grundblöcke

8.2.2. Inter Procedural

- **Laufzeit**

Programm	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
Time.tt	80	30	20	10
Mkeyes.tt	30	30	30	30
Darhttpd.tt	120	110	50	40
dc.tt	590	240	170	160
Concepts.tt	970	630	340	360
Joseki.tt	330	230	120	130
Gnuplot_x11.tt	4.210	770	500	510
Bc.tt	2.000	1.550	660	710
Nano.tt	1580	1.020	390	430
Gnugo.tt	52.840	23.210	23.260	23.690

Tabelle 8.2.2.1: Übersicht Laufzeit

- **Speicherverbrauch**

Programm	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
Time.tt	3	1	< 1	< 1
Mkeyes.tt	1	1	1	1
Darhttpd.tt	4	4	1	1
dc.tt	25	9	6	5
Concepts.tt	40	26	13	13
Joseki.tt	14	10	5	5
Gnuplot_x11.tt	183	32	20	20
Bc.tt	81	63	24	25
Nano.tt	67	42	14	15
Gnugo.tt	1.635	735	735	737

Tabelle 8.2.2.2: Übersicht Speicherverbrauch

- **Besuchte BBs**

Programm	Depth-first	Reverse Postorder	worklist	worklist rpo
Time.tt	2.362	1.067	621	575
Mkeyes.tt	1.256	1.751	647	533
Darhttpd.tt	5.355	4.363	2.085	2.086
dc.tt	19.402	7.385	5.189	5.098
Concepts.tt	10.323	7.812	5.306	5.506
Joseki.tt	9.213	6.716	3.785	4.133
Gnuplot_x11.tt	54.412	10.522	6.507	7.004
Bc.tt	31.094	17.861	10.926	11.754
Nano.tt	27.920	18.081	8.918	9.688
Gnugo.tt	381.612	325.501	189.400	190.325

Tabelle 8.2.2.3: Übersicht besuchte Grundblöcke

8.3. Post_Dom

8.3.1. Intra Procedural

- **Laufzeit**

Programm	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
Time.tt	60	60	60	60
Mkeyes.tt	100	100	90	90
Darhttpd.tt	150	150	140	140
dc.tt	500	510	500	510
Concepts.tt	240	260	250	250
Joseki.tt	1.150	1.200	1.160	1.180
Gnuplot_x11.tt	1.460	1.480	1.470	1.460
Bc.tt	1.280	1.280	1.300	1.280
Nano.tt	1.510	1.530	1.520	1.510
Gnugo.tt	232.820	233.830	232.350	231.930

Tabelle 8.3.1.1: Übersicht Laufzeit

- **Speicherverbrauch**

Programm	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
Time.tt	34	34	34	34
Mkeyes.tt	37	37	37	37
Darhttpd.tt	<1	<1	<1	<1
dc.tt	<1	<1	<1	<1
Concepts.tt	<1	<1	<1	<1
Joseki.tt	<1	<1	<1	<1
Gnuplot_x11.tt	<1	<1	<1	<1
Bc.tt	<1	<1	<1	<1
Nano.tt	<1	<1	<1	<1
Gnugo.tt	7	7	6	6

Tabelle 8.3.1.2: Übersicht Speicherverbrauch

- **Besuchte BBs**

Programm	Depth-first	Reverse Postorder	worklist	worklist rpo
Time.tt	1.875	1.068	548	430
Mkeyes.tt	803	1.917	764	328
Darhttpd.tt	2.990	3.761	1.612	1.385
dc.tt	10.235	4.980	2.250	2.691
Concepts.tt	6.220	6.625	3.206	2.840
Joseki.tt	16.790	14.729	7.343	7.209
Gnuplot_x11.tt	8.963	6.207	3.041	3.610
Bc.tt	10.772	13.550	6.858	4.714
Nano.tt	19.233	18.404	9.467	7.146
Gnugo.tt	200.567	237.932	156.569	78.460

Tabelle 8.3.1.3: Übersicht besuchte Grundblöcke

8.4. Dom

8.4.1. Intra Procedural

- **Laufzeit**

Programm	Depth-first (ms)	Reverse Postorder (ms)	worklist (ms)	worklist rpo (ms)
Time.tt	60	60	60	60
Mkeyes.tt	100	100	100	100
Darhttpd.tt	150	140	150	140
dc.tt	510	520	520	510
Concepts.tt	260	250	250	260
Joseki.tt	1.150	1.180	1.160	1.160
Gnuplot_x11.tt	1.470	1.490	1.460	1.470
Bc.tt	1.290	1.300	1.310	1.290
Nano.tt	1.500	1.540	1.510	1.540
Gnugo.tt	228.990	229.190	240.250	232.840

Tabelle 8.4.1.1: Übersicht Laufzeit

- **Speicherverbrauch**

Programm	Depth-first (MB)	Reverse Postorder (MB)	worklist (MB)	worklist rpo (MB)
Time.tt	34	34	34	34
Mkeyes.tt	36	36	36	36
Darhttpd.tt	<1	<1	<1	<1
dc.tt	<1	<1	<1	<1
Concepts.tt	<1	<1	<1	<1
Joseki.tt	<1	<1	<1	<1
Gnuplot_x11.tt	<1	<1	<1	<1
Bc.tt	<1	<1	<1	<1
Nano.tt	<1	<1	<1	<1
Gnugo.tt	9	10	10	9

Tabelle 8.4.1.2: Übersicht Speicherverbrauch

- **Besuchte BBs**

Programm	Depth-first	Reverse Postorder	worklist	worklist rpo
Time.tt	903	842	354	534
Mkeyes.tt	803	918	1012	1002
Darhttpd.tt	3.273	2.787	1.449	2.876
dc.tt	5.382	4.413	2.009	4.729
Concepts.tt	5.205	5.292	2.455	3.061
Joseki.tt	16.659	12.248	7.306	26.292
Gnuplot_x11.tt	9.186	5.654	2.853	8.232
Bc.tt	10.122	9.540	4.673	7.469
Nano.tt	16.585	13.344	9.451	23.785
Gnugo.tt	201.799	166.299	137.053	231.113

Tabelle 8.4.1.3: Übersicht besuchte Grundblöcke

9. Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben.
Ich habe keine anderen, als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet.
Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens.
Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht.
Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

_____

Ort, Datum, Unterschrift