Quellsprachorientiertes Debugging optimierter Programme*

Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Technische Universität München
Fakultät für Informatik

15.11.1994

* gefördert im Rahmen des SFB342 “Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen”, TP A1

Agenda

1. Motivation
2. Quellsprachorientiertes Debugging unoptimierter Programme
3. Probleme durch Optimierung
4. Lösungsansätze
5. Debugging optimierter Programme mit Hilfe von Datenflußanalysemethoden
6. Ergebnisse

Umfeld

Forschungsschwerpunkte am Lehrstuhl für Rechnertechnik und Rechnerorganisation:

1. Parallelrechner und ihre Programmierung
   - ⇝ Programmiermodelle und -umgebungen (4)
   - ⇝ Monitoring und Programmierwerkzeuge (10)
   - ⇝ Anwendungen (3)
2. Mikroprozessoren und Simulation
   - ⇝ Leistungsbewertung (2)
   - ⇝ Feinkörniger Parallelismus (1)
3. Technische Grundlagen (4)

Warum Fehlersuche in optimiertem Code?

Codetransformationen (Optimierungen) sind notwendig für:

- gute Performance auf RISC Architekturen
- parallele Architekturen (superskalar, VLIW, Mehrprozessorsysteme)

$\Rightarrow$ Produktionscode sollte optimiert sein!

Aber:

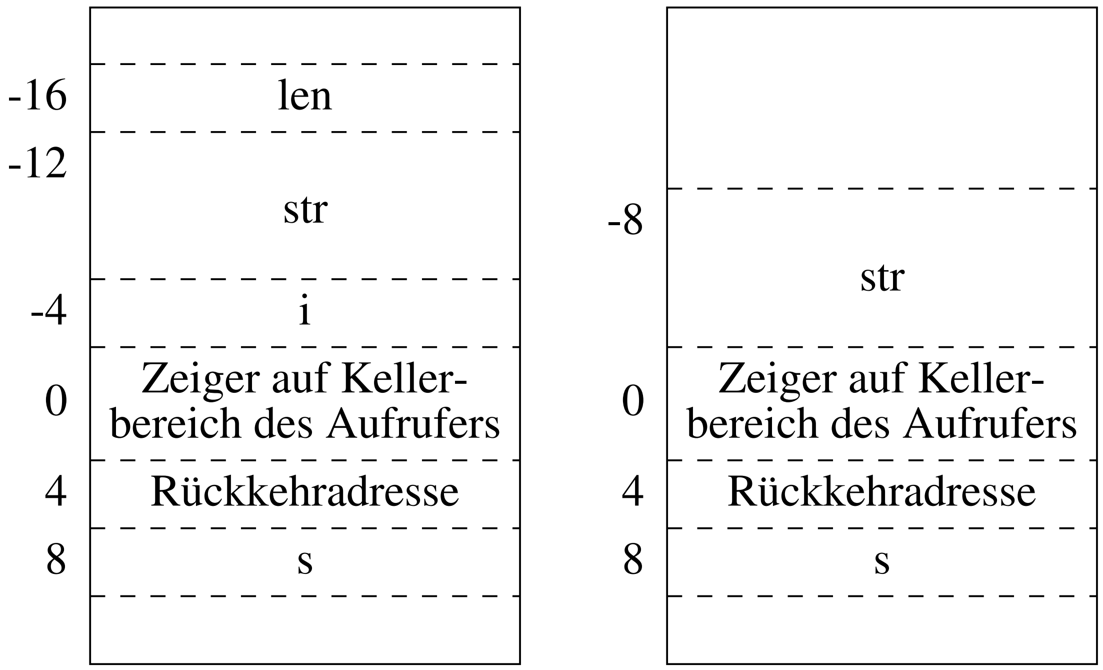
- Auch Produktionscode hat Fehler!
- Fehlersuche in unoptimierten Versionen oft nicht ausreichend:
  - $\leadsto$ Neuübersetzung wird notwendig
  - $\leadsto$ kein post-mortem Debugging möglich
  - $\leadsto$ Fehler muß reproduziert werden:
    sporadische Fehler, durch Optimierung “enttarnte” Fehler, Zeitbedingungen, Fehler in der Optimierung

Beispiel

Programmcode:

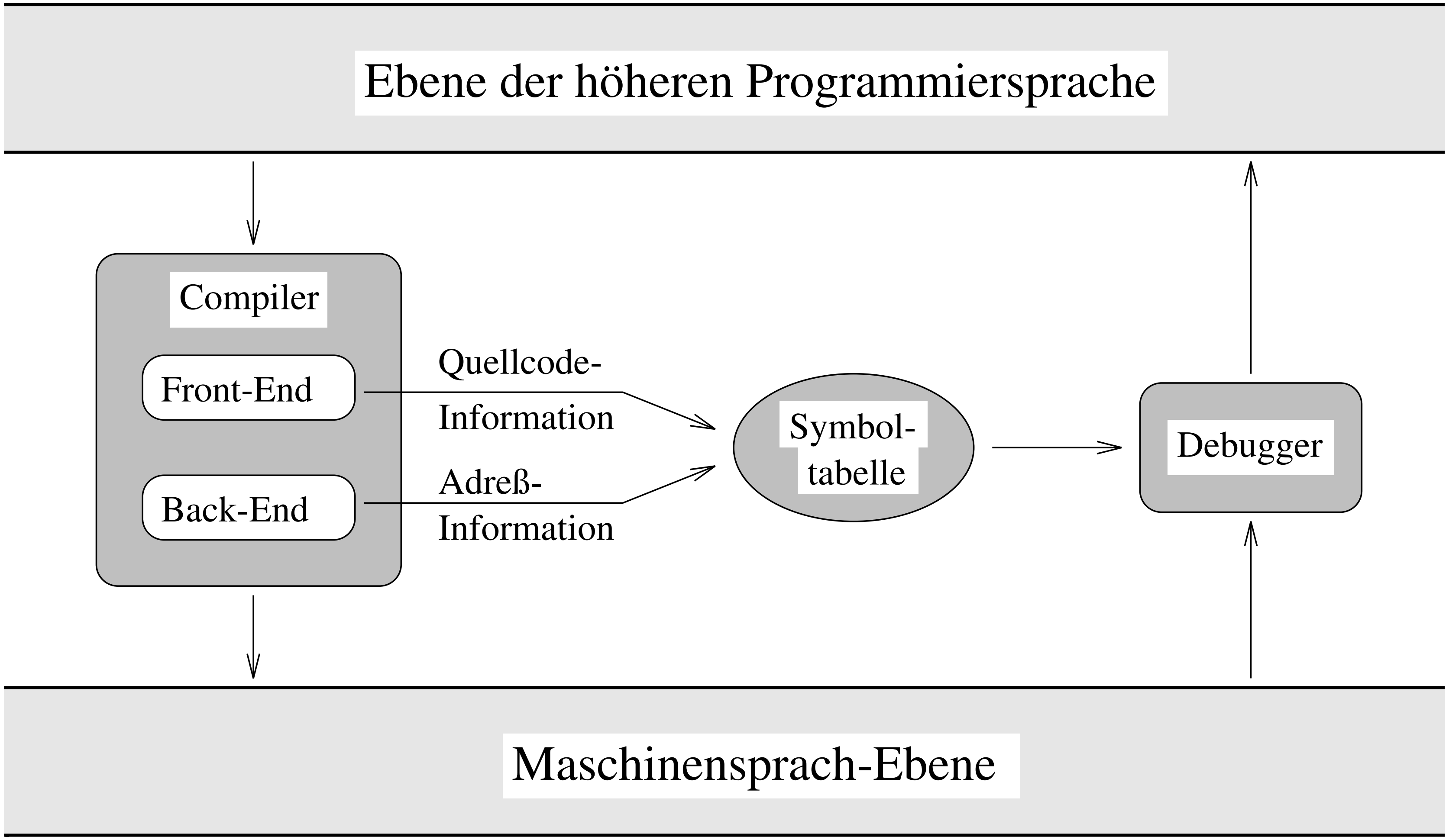
```
int foo(s)
char *s;
{
      int i;
      char str[8];
      int len;

      len = strlen(s);
      if (len > 8) {
            return(-1);  /* zu lang */
      }
      for (i=0; i<len; i++)
            str[i] = s[i];
      str[len] = '\0';
      /* ... (i nicht mehr benutzt) */
}
```

Kelleraufbau:



ohne Optimierung — mit Optimierung

Übersetzung und Debugging



Beispiel

```c
int a[10];
register short i;
int y;

if (a[i]>0) {
    y += a[i];
    i++;
}
. . .
```

```
0x8     sub   %fp,0x28,%o0       ! fetch &a
0xc     sll   %i5,0x2,%o1
0x10    ld    [%o0+%o1],%o2      ! fetch a[i]
0x14    tst   %02                ! if (a[i]>0) {
0x18    ble   0x3c
0x1c    nop

0x20    sub   %fp,0x28,%o0       !     fetch &a
0x24    sll   %i5,0x2,%o1
0x28    ld    [%o0+%o1],%o2      !     fetch a[i]
0x2c    ld    [%fp-0x2c],%o3     !     fetch y
0x30    add   %o3,%o2,%o3        !     y += a[i]
0x34    st    %o3,[%fp-0x2c]     !     store y

0x38    add   %i5,0x1,%i5        !     i++
                                 ! }
0x3c    . . .
```

Benutzung der Symboltabelleninformation

break 6

⇛ Instruktion an Adr. 0x20
durch Trap ersetzt

Zeilentabelle

Zeile	Adr.
5	0x8
6	0x20
7	0x38
9	0x3c

Variablentabelle

Var.	Typ	Ort	Klasse
a	int[10]	-0x28	lokal
i	short	i5	Register
y	int	-0x2c	lokal

Trap an Adr. 0x20 erreicht

⇛ stopped at line 6

print a

⇛ Auslesen Speicher zw. fp-0x28 . . . fp
Ausgabe als Folge von ganzen Zahlen

Probleme duch Optimierung

Annahmen herkömmlicher Debugger:

1. Umsetzung jeder Anweisung in zusammenhängende Instruktionssequenz
2. Relative Reihenfolge der Anweisungen und der zugehörigen Instruktionssequenzen identisch
3. 1-zu-1-Zuordnung zw. Anweisungen und erzeugten Instruktionssequenzen
4. 1-zu-1-Zuordnung zw. Variablen und Registern für deren gesamten Gültigkeitsbereich

Optimierungen verletzen diese Annahmen:

1. Instruktionsscheduling, Elimination gemeinsamer Teilausdrücke, ...
2. Verschiebung von schleifeninvariantem Code, Scheduling, Schleifenoptimierungen, ...
3. Elimination gemeinsamer Teilausdrücke, Elimination von passivem Code, ...
4. Registerzuteilung, *register promotion*, ...

```
...
for (i=0;i<10;i++) {
   if (a[i] > 0) {
      y = y + a[i]
      f = g + h
      w = a[i] * 2
      z = z + w
      x = a[i] + 1
   }
   else {
      w = a[i] * 2
      x = -a[i] + 1
      z = z - w
      y = y + a[i]
      f = g + h
   }
}
```

```
       ...
       mov   0,%i3           ! i = 0
       add   %o0,%i5,%l6     ! f = g + h
       add   %fp,-48,%i2     ! i2 = &a[i]
                             ! loop:
L1:    ld    [%i2],%i5       !    i5 = a[i]
       tst   %i5
       ble   L2
       sll   %i5,1,%i1       !    w = a[i] * 2
                             !    if (a[i] > 0)
       b     L3
       add   %i4,%i1,%i4     !       z = z + w
                             !    else
L2:    sub   %i4,%i1,%i4     !       z = z - w
L3:    inc   %i3             !    i++
       cmp   %i3,10
       add   %l7,%i5,%l7     !    y = y + a[i]
       bl    L1
       inc   4,%i2           !    i2 = &a[i]
                             !    if (i<10) goto loop
```

Debuggingprobleme im Beispiel

Setzen eines Haltepunkts auf Zeile 4:

- über herkömmliche Symboltabelle: Trap an Instruktion 2 (*semantic mapping*)
  $\Rightarrow$ Haltepunkt genau einmal statt 0 ... 10 mal erreicht
- Optimale Abbildung abhängig von Intention:
  - Verfolgung des Kontrollflusses: Instr. 10/11 (*syntactic mapping*)
  - Ermittlung des alten Werts von f: Instr. 2
  - Ermittlung des neuen Werts von y: Instr. 17 (falls i5$>$0)

Ermittlung von Variablenwerten (Haltepunkt Zeile 4, Instr. 10):

z: Wert korrekt (z ist aktuell)
y: Im Zielprogramm veralteter Wert (y nicht aktuell, *roll forward variable*)
w: Im Zielprogramm bereits neuer Wert (w nicht aktuell, *roll backward variable*)
f: Im ersten Schleifendurchlauf: f ist *roll backward variable*
In späteren Durchläufen: f ist aktuell

Forderungen an Debugger für optimierte Programme

Korrektes Verhalten

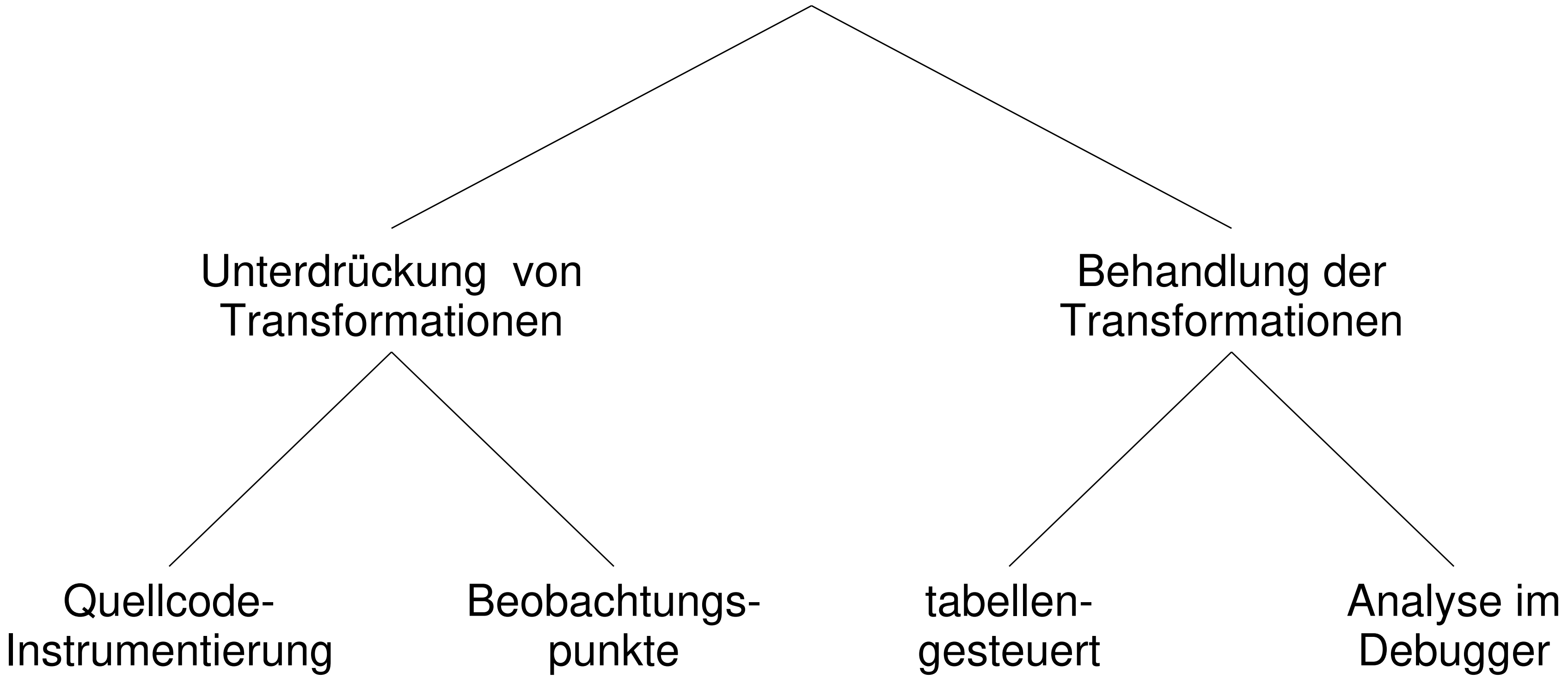
- Debugger zeigt nie ein gemäß Quellprogramm falsches oder irreführendes Verhalten (ohne entsprechenden Hinweis)
- Erklärung der durch Optimierung entstandenen Effekte auf Programmablauf und -zustand.

Transparentes Verhalten

- Debugger zeigt exakt dasselbe Verhalten wie bei unoptimiertem Programm
- In der Praxis ohne Einschränkung der Optimierung nicht erreichbar

$\Rightarrow$ Abschwächung: transparentes Verhalten, soweit möglich; in jedem Fall aber korrektes Verhalten

Methoden zum Debugging optimierter Programme



Quellcode-Instrumentierung

Einfügung von speziellen Debug-Anweisungen in den Quellcode, z.B. zur Ausgabe von Variablen oder zur Unterbrechung des Programms

$\Rightarrow$ Compiler stellt korrekte Ausführung der Anweisungen sicher

$\Rightarrow$ Problematische Optimierungen werden automatisch unterdrückt

$\Rightarrow$ Transparentes Debuggerverhalten

Problem: Bei Änderung der Instrumentierung ist Neuübersetzung notwendig

- Zeitaufwand
  $\Rightarrow$ inkrementelle Optimierung
- Keine Fortsetzung möglich
  $\Rightarrow$ kein sinnvolles interaktives Debugging

Beobachtungspunkte (*inspection points, interrupt points*)

Identifizierung der Stellen, an denen Debugger aufgerufen werden kann

- Einschränkung der Optimierung, so daß an diesen Stellen Zustand des Quellprogramms bestimmbar ist
- Falls erforderlich: Ersetzen einzelner Funktionen durch unoptimierten Code
  $\Rightarrow$ Fortsetzung möglich, da Zustand vollständig bestimmbar

$\Rightarrow$ Transparentes Debuggerverhalten

Problem: Für C und ähnliche Sprachen: fast jede Anweisung ist Beobachtungspunkt

$\Rightarrow$ Praktisch nur anweisungslokale Optimierung möglich

$\Rightarrow$ Anwendung bisher nur für streng objektorientierte Sprachen (TS, SELF)

Tabellengesteuerte Verfahren

Optimierer beschreibt *Effekte* der Optimierungen in erweiterter Symboltabelle

Beispiel: Bereiche (*ranges*)

Bereichstabelle

Var.	Adr.bereich	Ort	Klasse
i	0 ... 19	i3	Reg.
y	0 ... 8	l7	Reg.
w	0 ... 8	i1	Reg.
f	0 ... 2	l6	Reg.

Haltepunktabbildung

Zeile(n)	Adresse
1	1
2	5
3 ... 7	10
10 ... 14	13
16	14

$\Rightarrow$ Korrektes (transparentes) Debuggerverhalten

Probleme: Berechnung im Optimierer, feste Haltepunktabbildung, unflexibel

Analyse im Debugger

- Optimierer beschreibt Struktur des Quell- und Zielprogramms sowie die durch die Optimierung entstandene Abbildung
- Debugger analysiert Abbildung und bestimmt Effekte der Optimierungen

$\Rightarrow$ Korrektes (transparentes) Debuggerverhalten

Vorteile: Einfache Berechnung im Optimierer, sehr flexibel

Problem: Komplexe Analyse

$\Rightarrow$ Bisher nur für eingeschränkte Fälle realisiert
(lokale Optimierungen, keine Zeiger, feste Haltepunktabbildung)

Analyse zur Bestimmung von Variablenwerten

Aufgaben:

- Bestimmung des zu einer Variablen gehörigen Registers
- Feststellung, ob dieses Register den erwarteten Wert enthält

Anforderungen:

- Behandlung von Optimierungen
- Interaktives Debugging
- Unabhängigkeit von Optimiererdetails
- Einfache Realisierung im Optimierer
- Möglichst transparentes Verhalten (*recovery*, Laufzeitinformation)
- Beliebige Haltepunktabbildung
- Realistischer Einsatzbereich (globale Optimierungen, Zeiger)
- Formales Modell (Korrektheit)

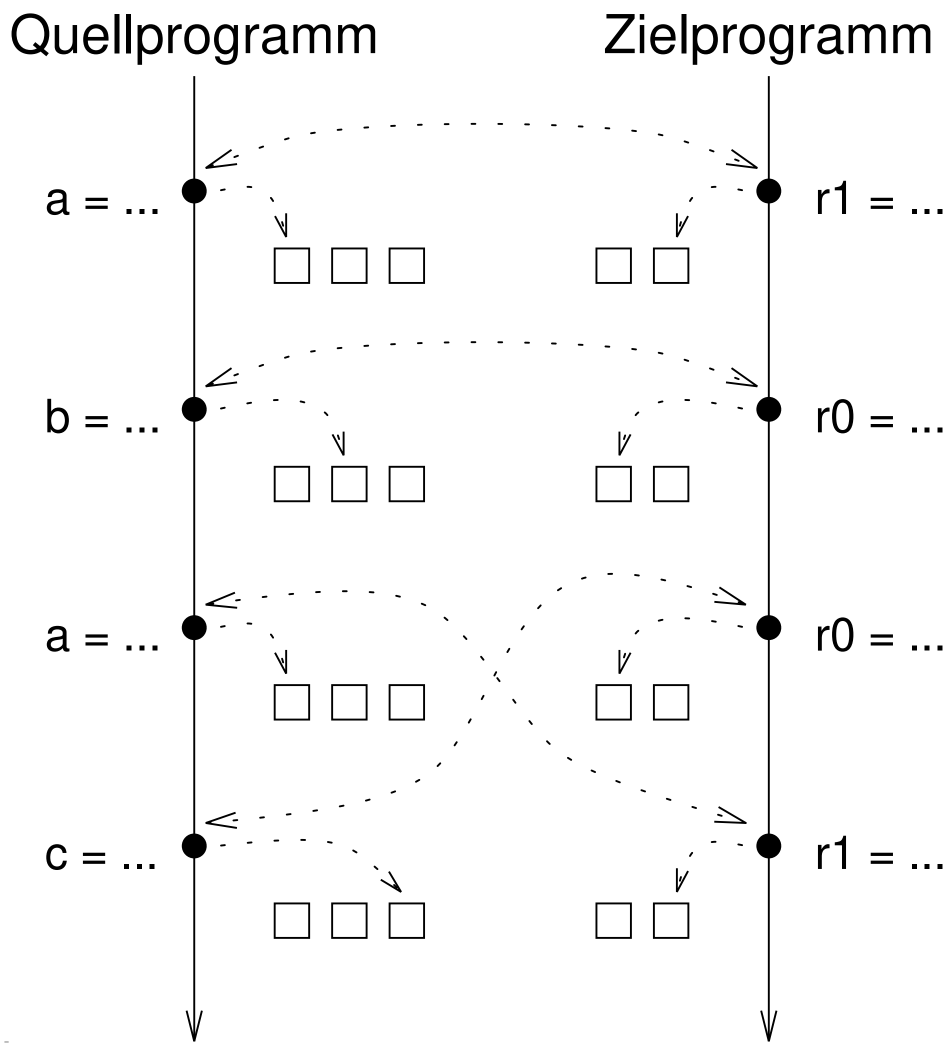
Formales Modell

Basis: Programmablauf gemäß operationeller Semantik:

- Folge von Zuständen
- Zustand = Variablenbelegung + Nachfolgeanweisung

Für compilierte Programme: Programmablauf*paar*:

- Ablauf des Quell- und Zielprogramms mit selber Eingabe
- Abbildung zwischen semantisch äquivalenten Operationen



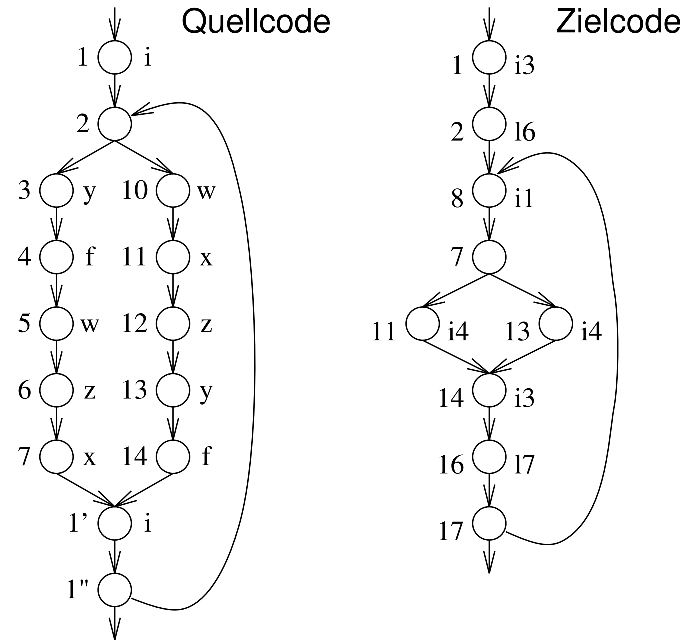
Schnittstelle Compiler-Debugger

Flußgraphen für Quell- und Zielcode

- Operationen, die Programmzustand ändern
- Modifizierte Variablen

Abbildungen zwischen Quell- und Zielcode

- Äquivalente Zuweisungen
  $\rightsquigarrow$ schreiben selben Wert
- Zusammengehörige Kanten
  $\rightsquigarrow$ aquivalente Kontrollflußsteuerung



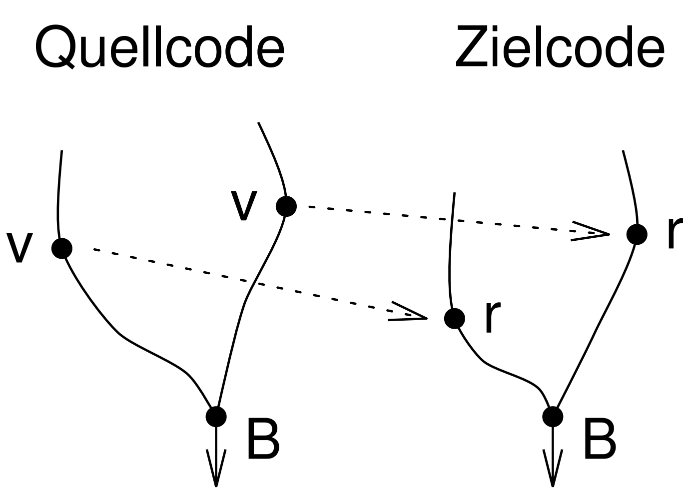
Source	Object
1	1
1’	14
3,13	16
4,14	2
5,10	8
6	11
12	13

Source	Object
<2,3>	<7,11>
<2,10>	<7,13>
<7,1’>	<11,14>
<14,1’>	<13,14>
<1,2>	<2,8>
<1",2>	<17,8>

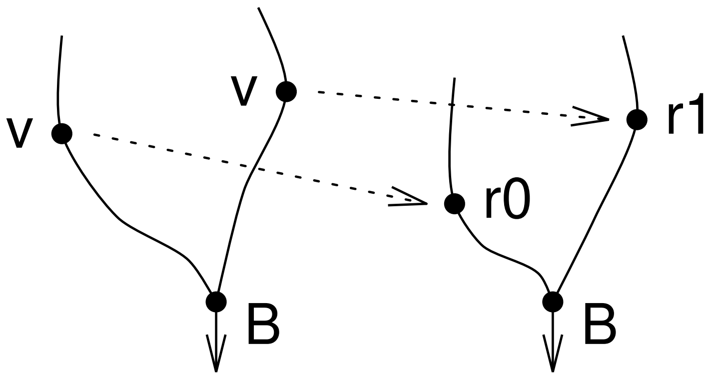
Bestimmung des zu einer Variablen gehörigen Registers

Betrachte alle erreichenden Definitionen von v:

- alle zugehörigen Speicherungen schreiben auf dasselbe Register r:
  - r ist das zu v gehörige Register



- Eine Definitionen hat keine zugehörige Speicherung, oder
- einige zugehörige Speicherungen schreiben auf ein anderes Register:
  - kein (eindeutiges) zugehöriges Register
  - Variable v ist nicht aktuell

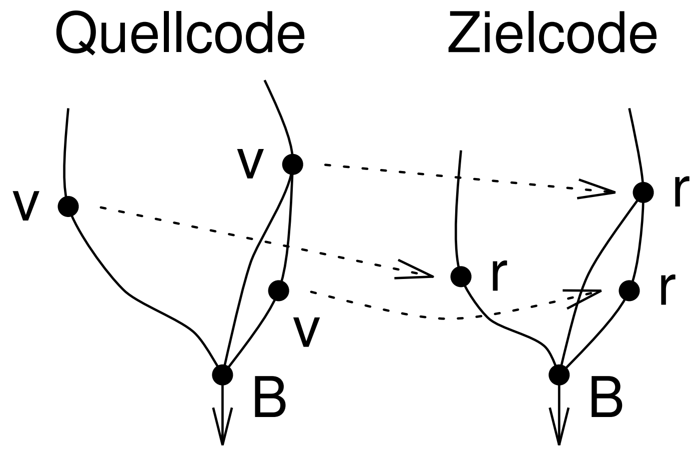


Prüfung der Aktualität

Betrachte alle erreichenden Definitionen von v:

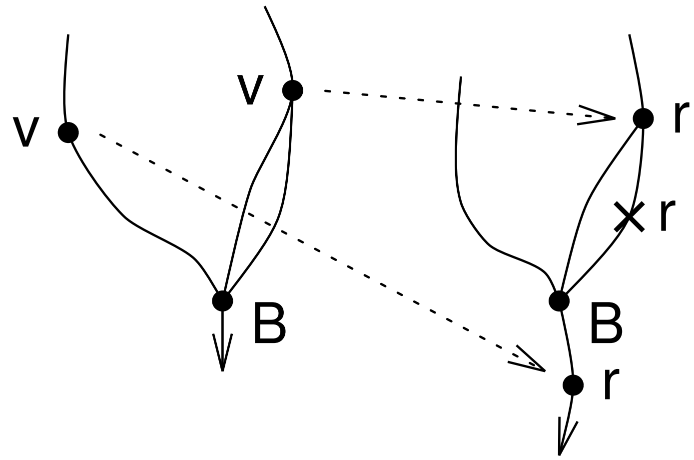
v ist aktuell in r, falls:

- alle zugehörigen Schreibzugriffe auf r erreichen den Haltepunkt im Zielprogramm entlang aller zulässigen Pfade



Falls v nicht aktuell in r ist, dann:

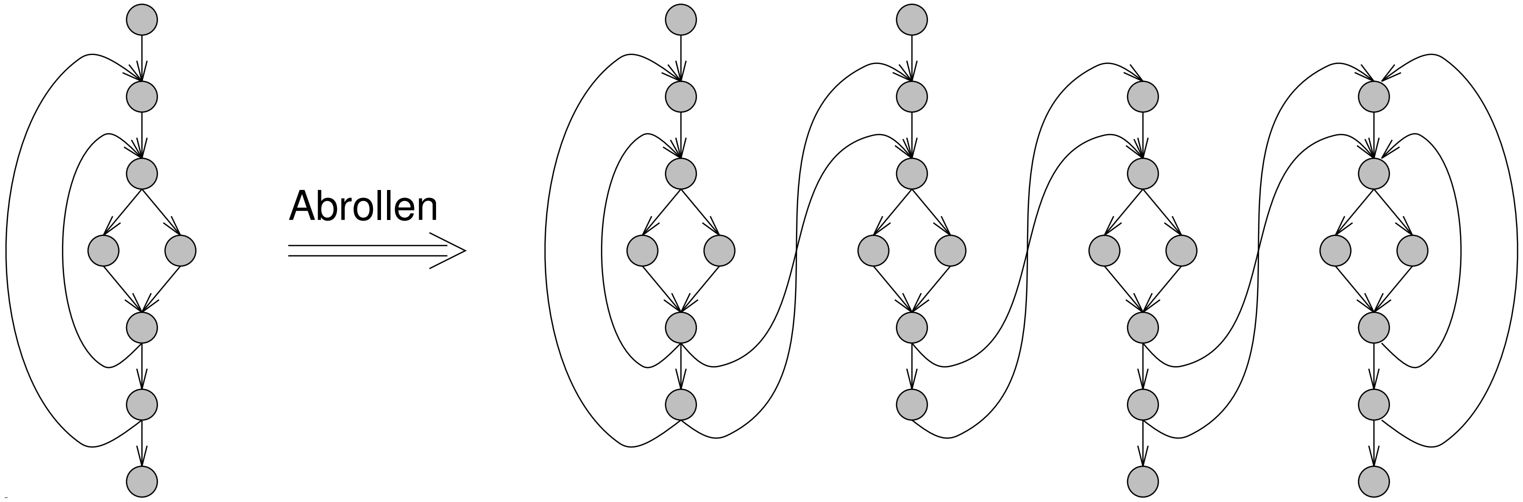
- eine zugehörige Speicherung erreicht den Haltepunkt im Zielprogramm entlang einiger zulässiger Pfade nicht



Zu lösende Probleme

- Einschränkung der Analyse auf die zulässigen Pfade
- Unterscheidung von Anweisungsinstanzen

⇒ Abgerollte Flußgraphen



- Terminierung des Verfahrens (unbeschränkt lange Pfade)

⇒ Obere Grenze, nur abhängig von Strukturparameter

Behandlung indirekter Zuweisungen

Bisheriger Ansatz noch nicht ausreichend:

Quellcode	Zielcode
1: a = x	1: a = x
2: *p = y	2: *p = y
3: · · ·	3: *q = z
4: *q = z	4: · · ·

a) Haltepunkt (3,3):

a aktuell, Instr. 1 kann Haltepunkt erreichen, muß aber nicht

b) Haltepunkt (4,4):

Instr. 1 und 2 können Haltepunkt erreichen, a ist nicht aktuell ϟ

Lösung: rekursive Analyse, betrachte nur letzte Definition

a) Haltepunkt (3,3):

1. Betrachte Def. 2 bei (3,3)
2. Betrachte Def. 1 bei (2,2)

$\Rightarrow$ a aktuell $\surd$

b) Haltepunkt (4,4):

1. Betrachte Def. 2 bei (4,4)

$\Rightarrow$ a nicht aktuell $\surd$

Algorithmus

1. Bestimme das der Variablen zugeordnete Register
2. Für $m=0,\ldots,4\cdot l+2$:

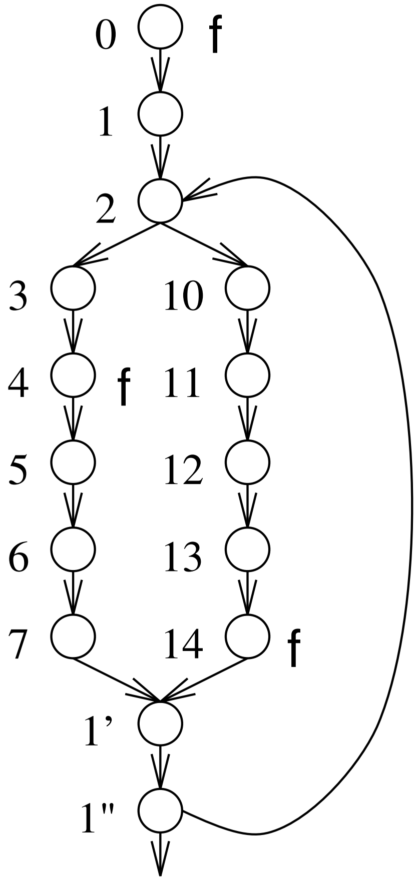
   Für jede Definition, die den Haltepunkt im Quellprogramm über m Rückwärtskanten erreicht:

   a) Rolle Flußgraphen ab
   b) Schränke Pfade im Zielflußgraph ein
   c) Bestimme Knoten im Zielflußgraph, die möglichen Instanzen des Haltepunkts entsprechen
   d) Bestimme Knoten im Zielflußgraph, die erwarteten Wert der Variablen generieren
   e) Prüfe, ob dieser Wert entlang aller verbliebenen Pfade im Zielflußgraph den Haltepunkt erreicht
3. Für alle indirekte Zuweisungen, die den Haltepunkt erreichen:

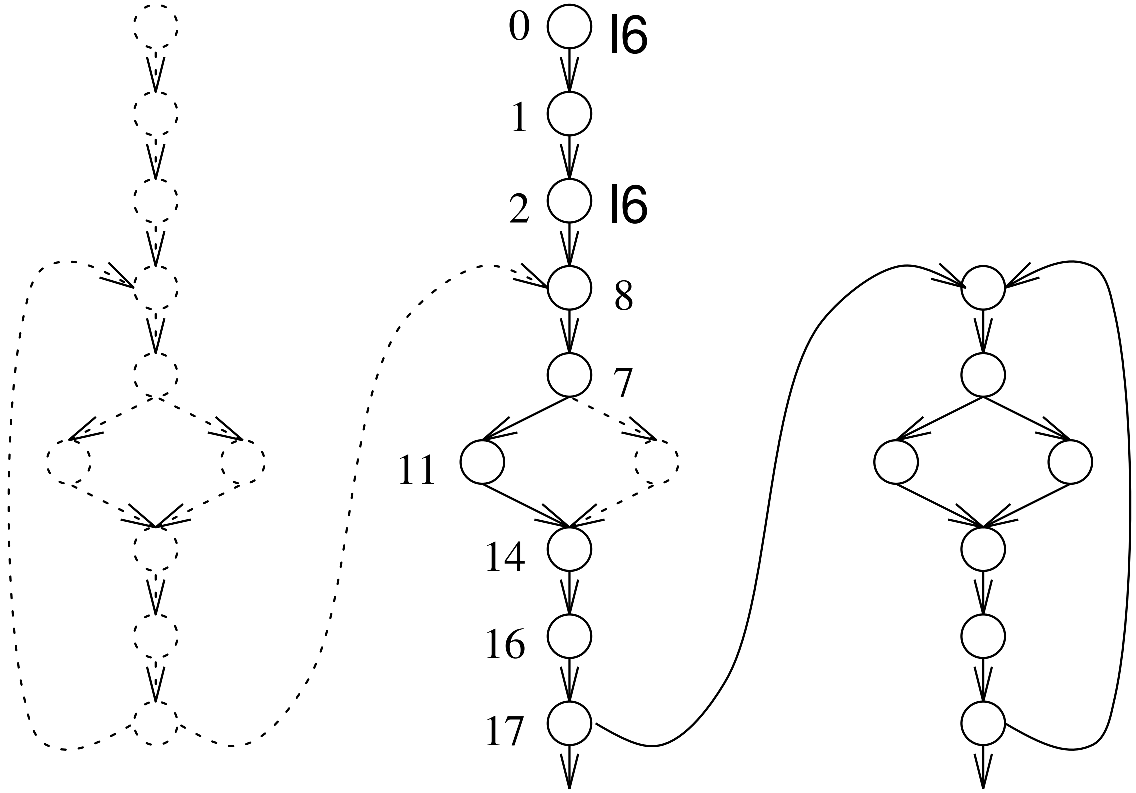
   Führe rekursive Analyse durch

Beispiel: Test der Definition 0 von Variable f

Quellflußgraph:



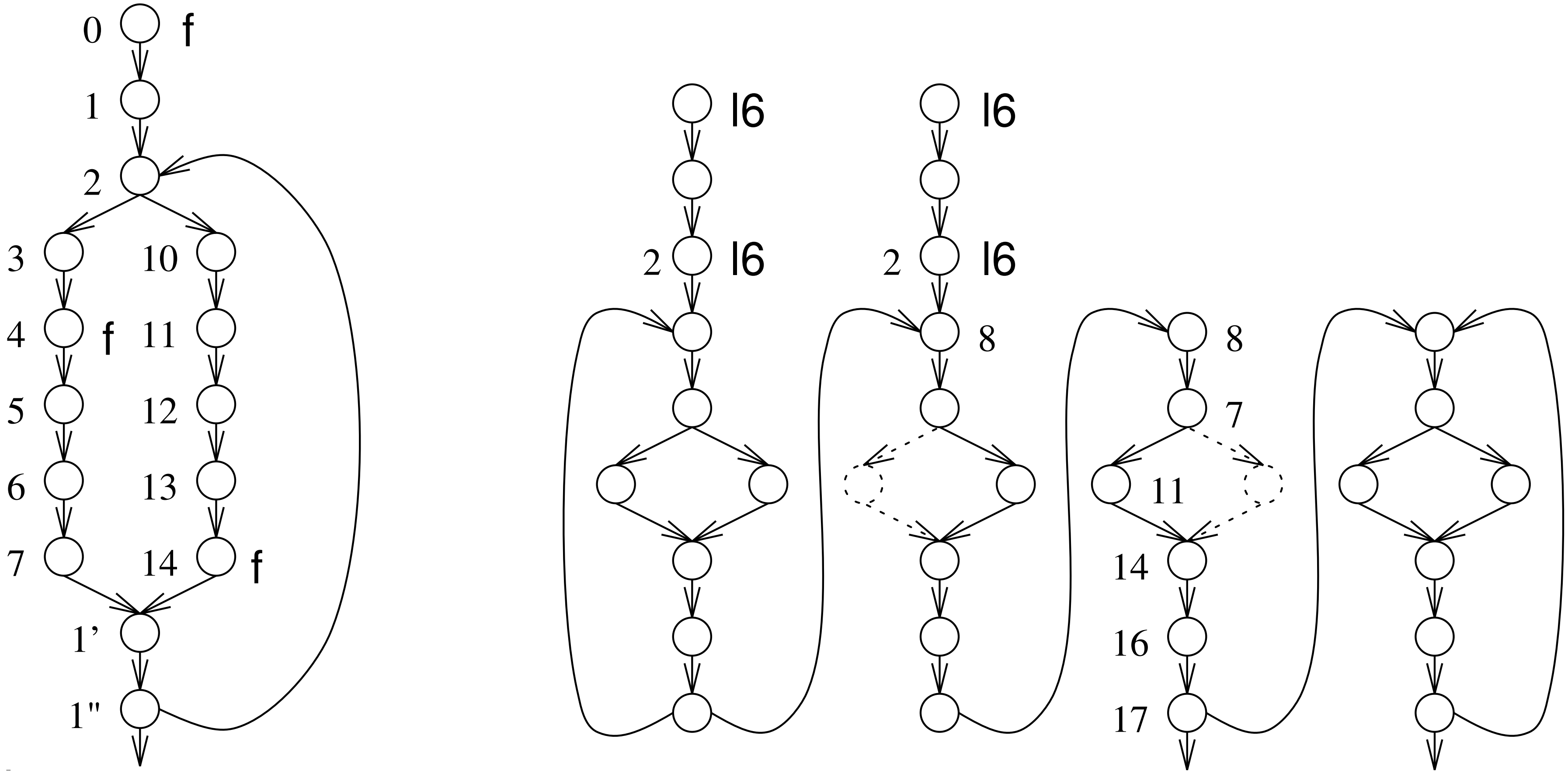
Abgerollter Zielflußgraph:



Beispiel: Test der Definition 14 von Variable f

Quellflußgraph:

Abgerollter Zielflußgraph:



Implementierung

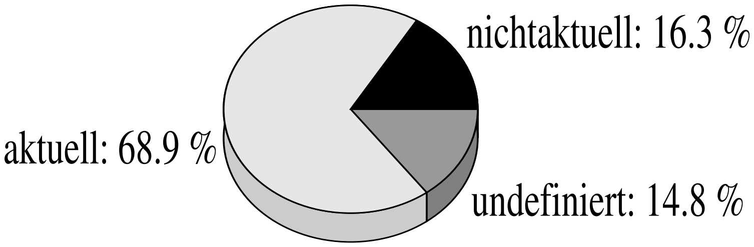
SUN C-Compiler für Sparc-Prozessor (SC1.0)

- Implementierungsaufwand:
  - $\leadsto$ Existierender Compiler: 94000 Codezeilen
  - $\leadsto$ Für Debugging: zusätzlich 2200 Codezeilen
  - $\leadsto$ Abbildung kann lokal in jedem Optimierungsschritt aktualisiert werden
- Mehraufwand für die Übersetzung mit Debugging-Unterstützung:
  - $\leadsto$ ohne Optimierung: +19% Laufzeit, +550% Dateigröße
  - $\leadsto$ mit Optimierung: +29% Laufzeit, +2000% Dateigröße

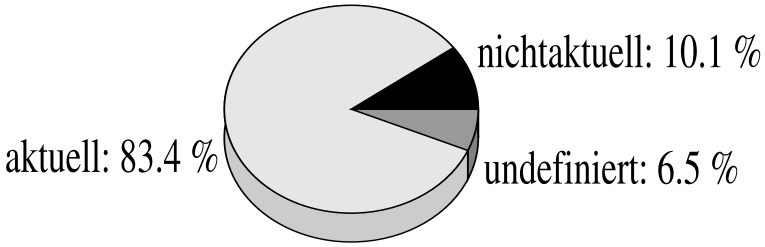
Analysealgorithmus

- Modul zur Bestimmung der Häufigkeit nicht aktueller Variablen
- Integration in parallelen Debugger DETOP

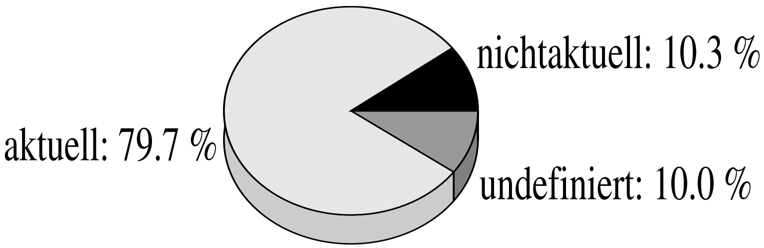
Häufigkeit nicht aktueller Variablen



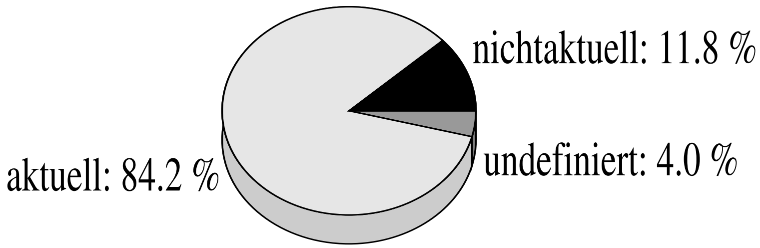
a) iropt, alle Variablen



c) iropt, referenzierte Variablen

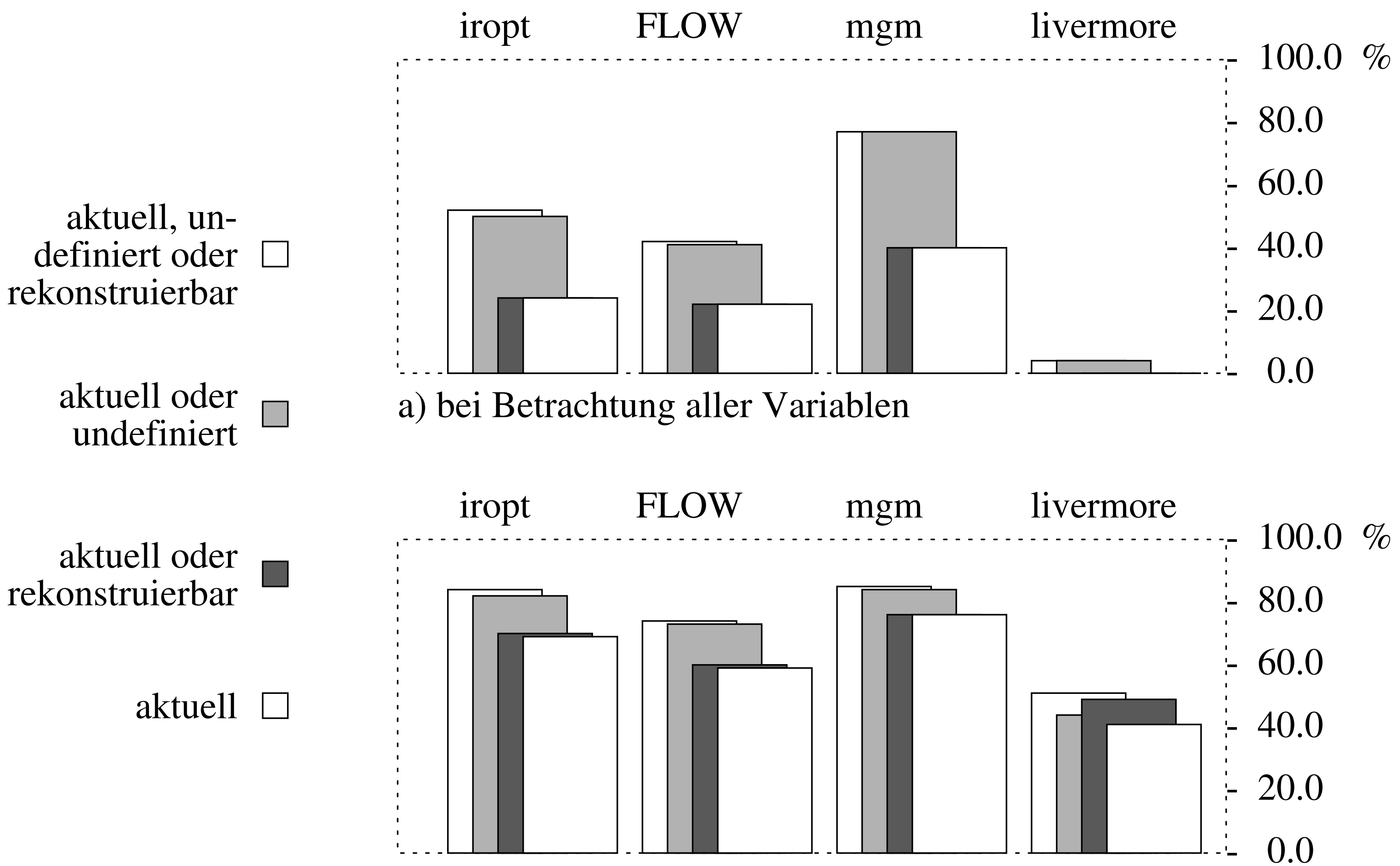


b) mgm, alle Variablen



d) mgm, referenzierte Variablen

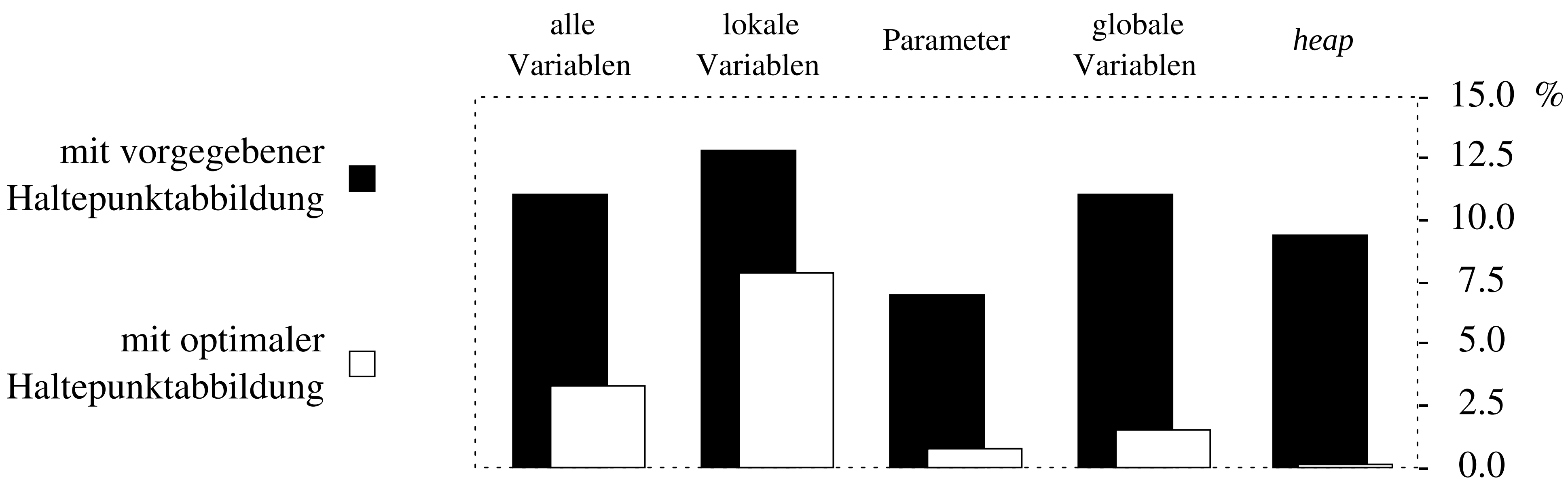
Haltepunkte mit voll transparentem Verhalten



a) bei Betrachtung aller Variablen

b) bei Betrachtung der referenzierten Variablen

Häufigkeit nicht aktueller Variablen bei optimaler Haltepunktabbildung



Fazit

- Debugging optimierter Programme wird die Ausnahme bleiben
- Debugging in Verbindung mit üblichen Zwischencode-Optimierungen praktikabel:
  - $\leadsto$ Analyse ist in allgemeiner Form möglich
  - $\leadsto$ Unterstützung im Compiler sehr einfach
  - $\leadsto$ Resultate zeigen, daß brauchbares Debuggerverhalten erzielt werden kann
- Bei Anwendung von Schleifentransformationen Analyse zwar theoretisch möglich, praktisch jedoch zu aufwendig