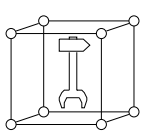
Quellcode-Debugging von global optimierten Programmen

Thomas Bemmerl, Roland Wismüller
Lehrstuhl für Rechnertechnik und Rechnerorganisation
Institut für Informatik
Technische Universität München
SFB 342, TP A1

bemmerl@.informatik.tu-muenchen.de
wismuell@.informatik.tu-muenchen.de

28.02.92



Motivation

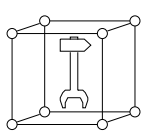
Compiler-Transformationen nötig für:

- Register- und Cache-Nutzung
- Vermeidung von Pipeline-Locks (superpipelined Prozessoren)
- Auslastung mehrerer Rechenwerke (superskalare Prozessoren, VLIW)
- Erhöhung der Parallelität in Schleifen (Multiprozessoren)

Konflikt zwischen Optimierung und Debugging

$\Rightarrow$ keine Optimierung während der Programmentwicklung
- schlechte Rechnernutzung

$\Rightarrow$ Neuübersetzung zum Debugging
- Compilierzeit
- Reproduzierbarkeit von Fehlern
- Realzeit-Programme



Probleme beim Debugging

Basis: Haltepunkt-basiertes Debugging

$\rightsquigarrow$ Setzen von Haltepunkten im Quellprogramm

$\rightsquigarrow$ Inspizieren des Programmzustands (Variable)

Beispiel:

S $\longrightarrow$

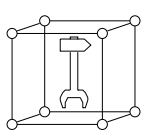
	Quellprogramm
1	**repeat**
2	a = b+c
3	b = b+d
4	c = f+h
5	d = b
6	**until** (b>99)

A $\longrightarrow$

	Zielprogramm
1	c=f+h
2	**repeat**
3	b,d = b+d
4	**until** (b>99)

Probleme beim Debugging:

- c hat bereits den Wert aus Zeile 4 (erster Schleifendurchlauf)
- a hat falschen Wert, da Anweisung 2 entfernt wurde
- in d steht schon das Resultat von Zeile 5



Lösungswege

Alternativen:

- Einschränkung der Optimierungen

  ⇒ mögliche Haltepunkte vor Übersetzung spezifiziert

  ⇒ Optimierungen, die Zustand an diesen Stellen verändern, unterbleiben

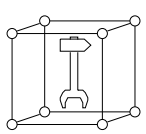
  ⇒ keine Änderung des Debuggers nötig

- Erweiterung des Debuggers

  ⇒ keine Einschränkung des Optimierers

  ⇒ Debugger muß Effekte der Optimierung erkennen bzw. behandeln

  ⇒ Compiler stellt dazu zusätzliche Information bereit



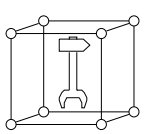
Anforderungen an Debugger

Ziele:

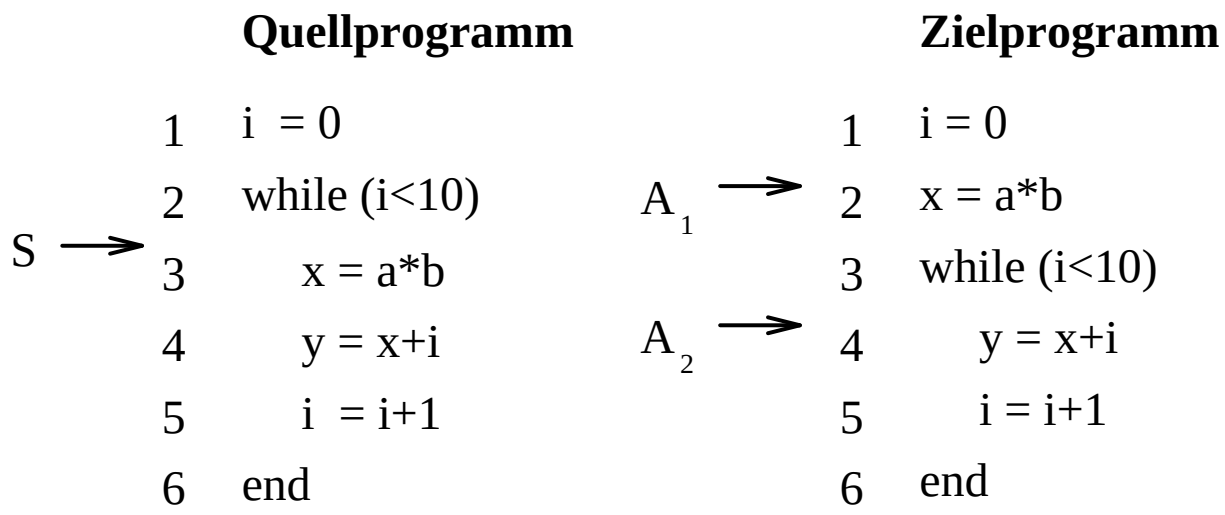
- $\rightsquigarrow$ korrektes Verhalten (*truthful behaviour*)
  - $\Rightarrow$ keine falschen bzw. irreführenden Ausgaben
  - $\Rightarrow$ Erkennung der Transformationen nötig
- $\rightsquigarrow$ transparentes Verhalten (*expected behaviour*)
  - $\Rightarrow$ Benutzer sieht Effekte der Optimierungen nicht
  - $\Rightarrow$ Behandlung der Transformationen nötig

Nötige Funktionen:

- $\rightsquigarrow$ Abbildung von Haltepunkten
- $\rightsquigarrow$ Bestimmung von Variableninhalten
- $\Rightarrow$ andere Funktionen (Aufrufkette, Einzelschritt, Modifikation von Variablen) hier nicht betrachtet



Bisherige Lösungsansätze (Haltepunkte)

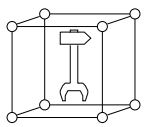


- *semantical mapping* $S \rightarrow A_1$
  Kriterium: Programmzustand
- *syntactical mapping* $S \rightarrow A_2$
  Kriterium: Position in Bezug auf Umgebung

$\Rightarrow$ zur Übersetzungszeit keine optimale Zuordnung möglich

[Zellweger]:

- Bestimmung des *syntactical mapping*
- Auswahl der Abbildung durch Benutzer



Bisherige Lösungsansätze (Variablen)

Programm sei an den Stellen S bzw. A angehalten:

- Variable V aktuell (*current*), wenn Zielcode-Variable existiert, deren Wert bei A immer mit dem von V bei S übereinstimmt
- Sonst ist V nicht aktuell (*noncurrent*)

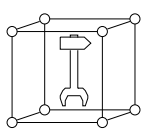
$\Rightarrow$ Für nicht aktuelle Variablen:
Versuch der Recovery

Beispiel:

Quellprogramm			**Zielprogramm**
1	a = e+c		
2	b = d+e		1 r0 = d+e
3	g = d+e		
4	a = 2*b	Haltepunkt	2 r1 = 2*r0

$\Rightarrow$ b,g aktuell (in Register $r0$)

$\Rightarrow$ a nicht aktuell,
Recovery durch Auswertung von $e + c$



Bisherige Lösungsansätze (Variablen)

[Hennessy]:

- Beschreibung der Methode für lokale Optimierungen

[Coutant,Meloy,Ruscetta]:

- Debugger DOC für optimierte C-Programme
- i.W. Behandlung von Registerzuteilung und Instruction-Scheduling

[Zurawski,Johnson]:

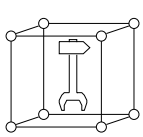
- Debugger für global optimierte Smalltalk-Programme (TS)
- Eingeschränkte Menge von Haltepunkten

Gemeinsamkeiten:

$\Rightarrow$ Bestimmung aktueller Variablen durch Compiler (i.a. zu pessimistisch)

$\Rightarrow$ feste Haltepunkt-Abbildung vorausgesetzt

$\Rightarrow$ Compiler/Debugger stark voneinander abhängig



Datenflußanalyse im Debugger

Effekte der Optimierung durch Daten- und Programmflußanalyse im Debugger bestimmbar

Vom Compiler benötigt:

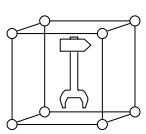
- Repräsentation (Flußgraphen/Syntaxbäume) von Quell- und Zielprogramm
- Abbildung $code$, ordnet jeder Anweisung im Quellprogramm die Anweisung im Zielprogramm zu, die die äquivalente Zustandsänderung vornimmt.

$\Rightarrow$ Analyse erst bei Bedarf

$\Rightarrow$ Berücksichtigung von Laufzeitinformation möglich

$\Rightarrow$ große Klasse von Optimierungen einheitlich zu behandeln

$\Rightarrow$ gekoppelte Analyse von zwei Programmen



Bestimmung aktueller Variablen

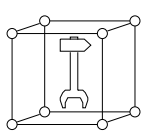
- Bestimmung von $RD(S, V)$
  (mögliche Definitionen von V im Quellprogramm, Startknoten für Initialisierung)
- V aktuell, falls

  (1) alle $I = code(D)$ mit $D \in RD(S, V) - \{Start\}$ definieren die selbe Zielcode-Variable V'
  (die V zugeordnete Varaible ist eindeutig bestimmbar)

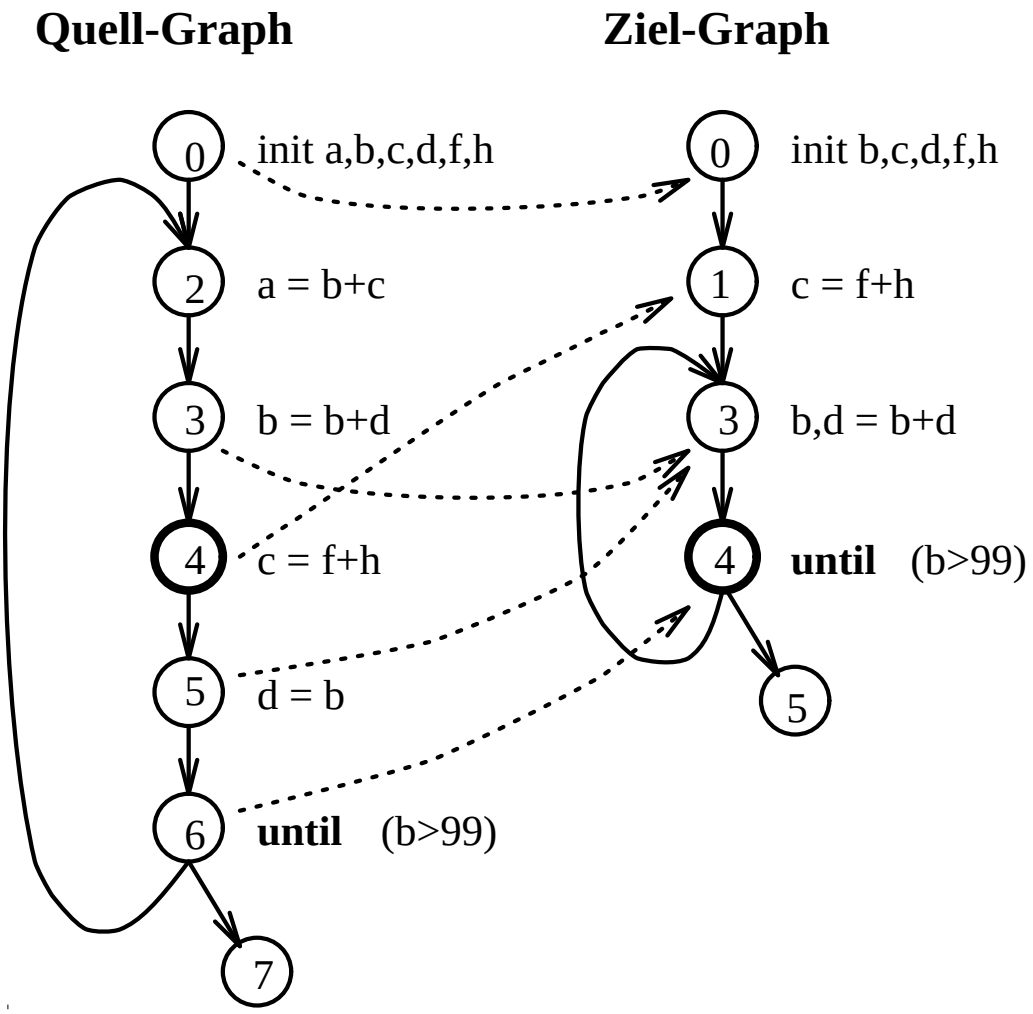
  (2) f.a. $D \in RD(S, V) : \ I = code(D)$ ist in jedem Programmablauf bei A verfügbar, in dem D die Stelle S erreicht ($I \in AD(A, V')$).
  (der Wert von V' bei A stimmt immer mit dem Wert von V bei S überein)

$\Rightarrow$ Recovery möglich, falls $|RD(S, V)| = 1$

$\Rightarrow$ Bestimmung von $AD(A, V')$ ist ein Datenfluß-problem mit Nebenbedingungen (*qualified dataflow problem*)



Beispiel



Für beliebigen Schleifendurchlauf:

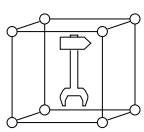
$$RD(4, c) = \{0, 4\} \qquad AD(4, c) = \{1\}$$

Im ersten Durchlauf:

$$RD(4, c) = \{0\} \qquad AD(4, c) = \{1\}$$

In allen weiteren Durchläufen:

$$RD(4, c) = \{4\} \qquad AD(4, c) = \{1\}$$



Haltepunkt-Zuordnung

Optimaler Haltepunkt:

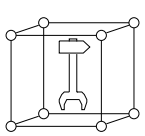
- „semantischer“ Aspekt
  (2) $\Rightarrow$ Stelle A im Zielcode, an der möglichst viele Variablen (aus einer gegebenen Menge) aktuell sind.
- „syntaktischer“ Aspekt
  Haltepunkt A_s wird genau dann erreicht, wenn S im Quellprogramm erreicht wird.

Ideal: beide Aspekte
z.B. falls A syntaktischen Aspekt nicht erfüllt, aber A_s postdominiert:

A_s: as_reached = TRUE
führe Programm fort

A: **if** (as_reached)
then halte Programm an
else führe Programm fort

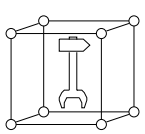
$\Rightarrow$ andere Fälle ähnlich (evtl. mit Logging)



Implementierung

Implementierung

- ⇝ C-Compiler für Sun-Sparc
- ⇝ Debugger DETOP der Werkzeugumgebung TOPSYS
- ⇝ Testversion der Datenflußanalyse

weitere Forschung

- ⇝ Aufwandsmessungen, Optimierung
- ⇝ Bestimmung des *syntactic mapping*
- ⇝ optimale Haltepunkte
- ⇝ automatisches Logging von Variablen
- ⇝ Kombination mit anderen Techniken (*slicing, reverse execution, flowback analysis*)
- ⇝ Schleifenoptimierungen, Parallelisierung