

Compiler II: Skalare Optimierung

Andreas Koch

FG Eingebettete Systeme und ihre Anwendungen
Informatik, TU Darmstadt

Sommersemester 2012

Dead Code Elimination

- Nutzloser Code
 - Keine weitere Operation verwendet Ergebnis
 - Genauer: Eine weitere Verwendung des Ergebnisses ist von außen nicht sichtbar
- Unerreichbarer Code
 - Kann auf keinem Pfad im CFG erreicht werden

Hier: Konzentration auf Entfernung nutzlosen Codes

Dead Code Elimination

Kritische Operationen haben nach außen sichtbare Effekte

- Müssen immer ausgeführt werden
- Return-Anweisungen
- Zuweisungen an var-Parameter, globale und nicht-lokale Variablen
- Unterprogrammaufrufe (wenn keine IPO vorhanden)
- Ein-Ausgabe-Anweisungen

Für VL vereinfacht: Nur Ausgabeoperationen relevant

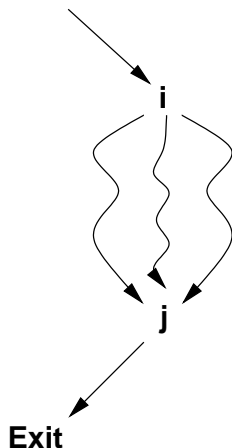
- Markieren benötigter Operationen
 - Markiere **kritische** Operationen
 - Untersuche deren Operanden und markiere die zugehörigen Definitionen als benötigt
 - Solange noch weitere benötigte Operationen dazu kommen: Wiederholen
- Entfernen toter Operationen
 - Alle nicht markierten Operationen entfernen

➔ Klassisches *Mark-and-Sweep* Vorgehen

- Für die meisten Operationen einfach
- Was bei Kontrollfluß (Kanten) zwischen Blöcken?
- Gleiche Grundidee wie bei anderen Anweisungen
 - Unbedingte Sprünge werden immer benötigt
 - Ausführung muß ja weitergehen
 - Bedingte Sprunganweisung: genauer ansehen
 - Ein Zweig wird **nur** benötigt, wenn er mindestens zu einer benötigten Anweisung führt
- Vorgehensweise
 - Bei Markieren einer Anweisung auch gleich **entscheidende** Verzweigung mitmarkieren
 - Leicht gesagt, aber wie genau diese Verzweigung finden?

Ein Knoten j **postdominiert** den Knoten i in einem CFG, wenn alle Pfade von i zum Endknoten des CFG durch den Knoten j führen.

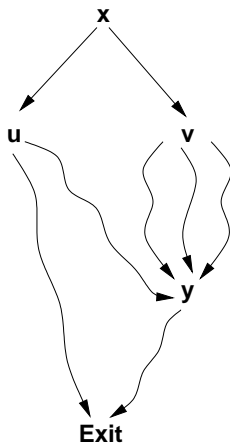
A. Koch



y ist von x **kontrollabhängig** genau dann, wenn

- 1 es einen nicht-leeren Pfad von x zu y gibt und jeder auf diesem Pfad liegende Knoten von y postdominiert wird,
- 2 x aber nicht strikt von y postdominiert wird.

- y postdominiert v und alle Knoten dazwischen
- y postdominiert nicht x
- y ist von x **kontrollabhängig**



Andere Deutung

- Zwei oder mehr Kanten verlassen Block x
- Nach Eintritt in eine der Kanten wird y in jeden Fall ausgeführt
- Über die andere(n) Kante(n) kann der Endknoten ohne y erreicht werden

Damit entscheidet Bedingung am Ende von x , ob y ausgeführt wird.

➔ Wenn Anweisung in y benötigt wird, wird damit auch die Entscheidung in x benötigt

Wie Kontrollabhängigkeit berechnen?

Hat etwas mit **Postdominanz** zu tun.

A. Koch

Zusammenhang:

- Postdominanz im CFG
- \leftrightarrow Dominanz im **umgekehrten** CFG
 - Richtung der Kanten vertauscht
 - *reversed CFG* (rCFG)

➔ Dominanzberechnung bekannt (Brandis & Mössenböck)

Reicht aber noch nicht ganz: Wo genau ist der y **nahegelegenste** Punkt, an dem die Entscheidung fällt?

➔ Wo ist der y nahegelegenste Knoten, bei dem auch eine Abzweigung an y **vorbei** genommen werden kann?

Analoge Betrachtung bei Dominatoren:

Welche Knoten w liegen **gerade außerhalb** der Dominanz eines Knotens x ?

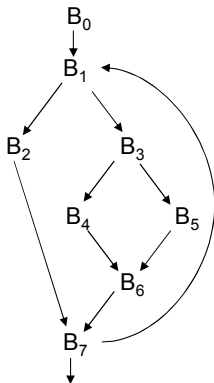
A. Koch

Anders: An welchem Knoten w kann aus dessen Ausführung nicht mehr sicher auf die vorherige Ausführung von x geschlossen werden?

Dominatorgrenze $DF(x)$

Knoten w , bei denen ein Vorgänger q durch x dominiert wird ($q \in \text{preds}(w) \wedge x \in \text{DOM}(q)$), aber w selbst nicht von x strikt dominiert ist ($x \notin \text{DOM}(w) - \{w\}$), heissen die **Dominatorgrenze** von x , mit $w \in DF(x)$.

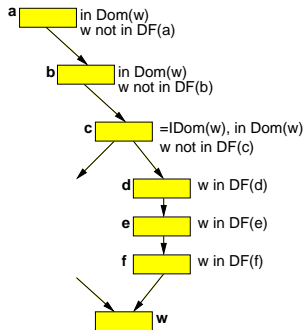
Beispiel Dominatorgrenze



	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

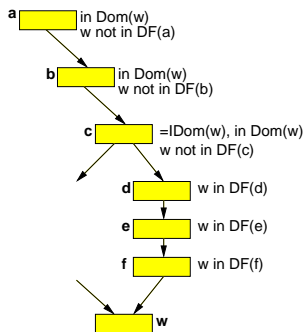
Berechnung der Dominatorgrenzen - Beobachtung

- Knoten auf Dominatorgrenze sind immer Merge-Knoten
- Vorgänger x eines Merge-Knotens w haben $w \in DF(x)$, wenn gilt $x \notin \text{DOM}(w)$
- Dominatoren z der Vorgänger x von w haben auch $w \in DF(z)$, wenn gilt $z \notin \text{DOM}(w)$



Berechnung von Dominatororgrenzen - Vorgehensweise

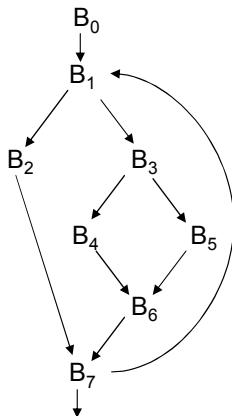
- 1 Finde Merge-Points als w
- 2 Beginne Untersuchung bei direkten Vorgängern x des Merge-Points w
- 3 Klettere rückwärts weiter via IDOM des aktuellen Knotens x
 - Setze $DF(x) = DF(x) \cup \{w\}$, bis $x = \text{IDOM}(w)$



Berechnung von Dominatorgrenzen - Algorithmus

```
foreach node n in CFG do  
  DF(n) :=  $\emptyset$   
foreach node n in CFG do  
  if |preds(n)| > 1 then  
    foreach p in preds(n) do  
      runner := p  
      while runner  $\neq$  IDOM(n)  $\wedge$  runner  $\neq$  n do  
        DF(runner) := DF(runner)  $\cup$  { n }  
        runner := IDOM(runner)
```

- 1 Bearbeite B6: Zu B5, dort B6 in DF(B5), Ende bei B3. Zu B4, dort B6 in DF(B4), Ende bei B3.
- 2 Bearbeite B7: Zu B2, dort B7 in DF(B2), Ende bei B1. Zu B6, dort B7 in DF(B6), zu B3, dort B7 in DF(B3), Ende bei B1.
- 3 Bearbeite B1: Zu B0, dort Ende. Zu B7, dort B1 in DF(B7), dort Ende.



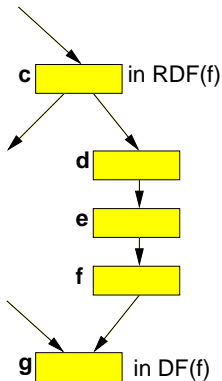
	0	1	2	3	4	5	6	7
DOM	0	0,1	0,1,2	0,1,3	0,1,3,4	0,1,3,5	0,1,3,6	0,1,7
DF	-	-	7	7	6	6	7	1

... zurück zu Dead Code Elimination

Gesucht: Verzweigungen, von denen benötigte Anweisung i
kontrollabhängig ist

A. Koch

➡ Sind Dominatorgrenzen von $\text{block}(i)$ im reversen CFG:
 $\text{RDF}(\text{block}(i))$



Markiere benötigte Operationen

MarkPass

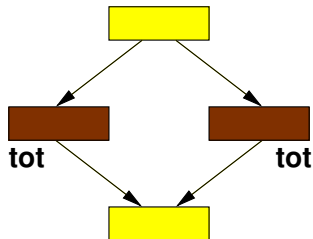
```
foreach op i
  clear i's mark
  if i is critical then
    mark i
    add i to WorkList
while (Worklist !=  $\emptyset$ )
  remove i from WorkList
    (i has form "x - y op z")
  if def(y) is not marked then
    mark def(y)
    add def(y) to WorkList
  if def(z) is not marked then
    mark def(z)
    add def(z) to WorkList
  foreach b  $\in$  RDF(block(i))
    mark the block-ending
      branch j in b
    add j to WorkList
```

A. Koch

```
Sweep
  foreach op i
    if i is not marked then
      if i is a branch then
        rewrite with a jump to i's nearest useful post-dominator
      if i is not a jump then
        delete i
```

- Lösche unmarkierte Operationen
- “Verbiege” unmarkierte Verzweigung
 - Setze Ausführung bei nächstgelegenen Postdominator mit nützlichen Operationen fort

Beispiel Verbiegen von Verzweigungen 1

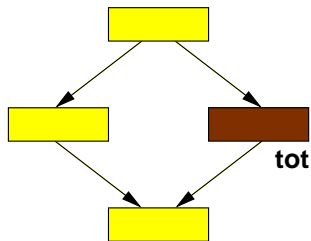


Vorher

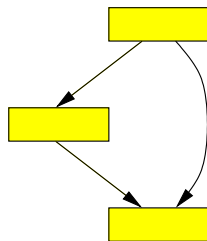


Nachher

Beispiel Verbiegen von Verzweigungen 2

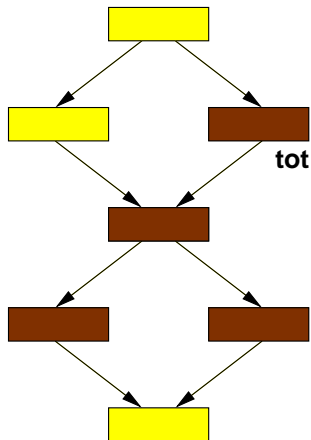


Vorher

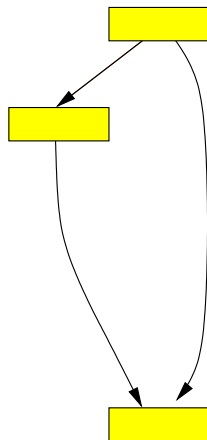


Nachher

Beispiel Verbiegen von Verzweigungen 3



Vorher



Nachher

- Gesamter Ablauf von Dead():
 - 1 MarkPass()
 - 2 SweepPass()
- Kann leere Blöcke hinterlassen
- Aufräumen mit nächstem Algorithmus

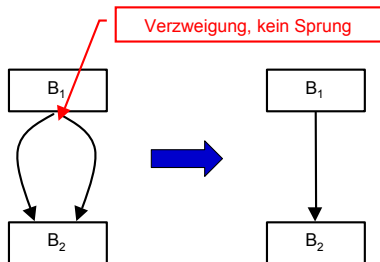
Bereinigen des CFG

- Nach Optimierung kann CFG leere Blöcke enthalten
- Leere Blöcke enden mit Übergang zum nächsten Block
 - Unbedingter Sprung (ein Nachfolger)
 - Bedingte Sprünge für Verzweigungen
- Kann zu Sprung-zu-Sprung führen (langsam & platzverschwendend)
- Beseitigen!

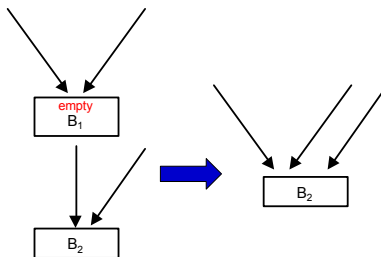
➡ Algorithmus CLEAN: Vier Schritte

Entferne redundante Verzweigung

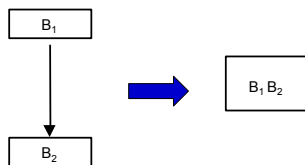
- Entsteht durch:
Verbiegen von
Verzweigungen
- Vorgehen: Ersetze
Verzweigung durch
Sprung



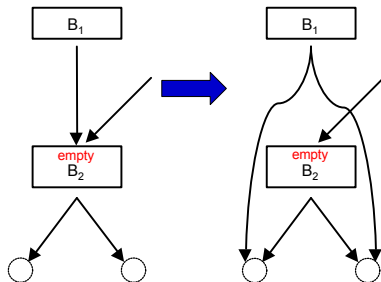
- Entsteht durch:
Gelöschte Operationen
in B1
- Voraussetzungen:
Leerer Block B1 endet
mit Sprung
- Vorgehen:
 - Verbiege eingehende
Kanten von B1 zu B2
 - Entferne B1



- Entsteht durch: Vereinfachte Kanten aus B1
- Voraussetzungen
 - B1 endet mit einem unbedingten Sprung
 - B2 hat genau einen Vorgänger
- Vorgehen:
 - Verschmelze beide Blöcke
 - Entferne damit den Sprung



- Entsteht durch:
Gelöschte Operationen
in B2
- Voraussetzungen
 - B1 endet mit Sprung
 - B2 ist leer und endet
mit Verzweigung
- Vorgehen:
 - Kopiere Verzweigung
von B2 ans Ende von
B1
 - Kann B2 unerreichbar
machen



- 1 Bearbeite Blöcke in **postorder**
 - Nachfolger eines Blockes b vor b selber bearbeiten
- 2 An jedem Block feste Abarbeitungsreihenfolge
 - 1 Entferne redundante Verzweigungen
 - Entfernt Kante, erzeugt neuen Sprung
 - 2 Beseitige leere Blöcke
 - Entfernt Knoten
 - 3 Verschmelze Blöcke
 - Entfernt Knoten und Kante
 - 4 Ziehe Verzweigungen heraus
 - Fügt neue Kante hinzu
- 3 Mehrere Durchgänge erforderlich
 - Postorder-Reihenfolge nach jedem Durchgang neu berechnen

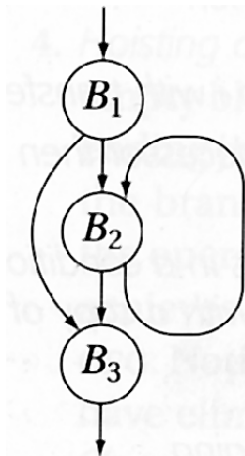
```
CleanPass()
  foreach block i, in postorder
    if i ends in a branch then
      if both targets are identical then
        rewrite with a jump
    if i ends in a jump to j then
      if i is empty then
        merge i with j
      else if j has only one predecessor
        merge i with j
      else if j is empty & j has a branch then
        rewrite i's jump with j's branch

Clean()
  until CFG stops changing
  compute postorder
  CleanPass()
```

Beispiel: Leere Schleife (**B2** leer)

CLEAN alleine kann **B2** nicht beseitigen

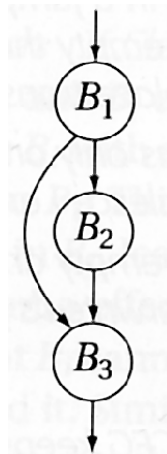
- Verzweigung am Ende von **B2** hat verschiedene Ziele
 - Ist nicht redundant, kann nicht in Sprung konvertiert werden
- **B2** endet nicht mit einem Sprung
 - Kein Zusammenfassen mit **B3**
- Vorgänger **B1** von **B2** endet mit Verzweigung
 - Kein Zusammenfassen von **B1** mit **B2**



A. Koch

➡ Geht aber in Kooperation mit DEAD!

- Ergebnis von DEAD:
- **B1**, **B3** enthalten nützliche Operationen
- **B2** nicht: Verzweigung (**B2**, **B2**) nutzlos
 - $B2 \notin RDF(B3)$
- Bedingungs-berechnung für Verzweigung nutzlos
- Verzweigung (**B2**, **B2**) in Sprung zu nützlichem Postdominator von **B2** umwandeln

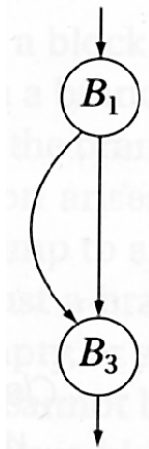


A. Koch

Vorgehen von CLEAN nach DEAD 1

- **B2** endet mit Sprung zu **B3** und ist selbst leer
- Ändere Sprung (**B1,B2**) zu (**B1,B3**)
 - Entfernt **B2**

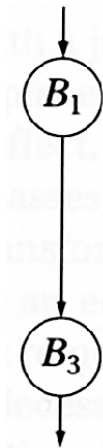
A. Koch



Vorgehen von CLEAN nach DEAD 2

- Verzweigung in **B1** redundant:
Umschreiben zu Sprung
- Abschliessend: **B3** mit einzigem
Vorgänger verschmelzen

A. Koch



Spezialisierung

- Üblicherweise: Erzeuge Code, der im Allgemeinfall funktioniert
- Wenn beweisbar, dass nicht alle Fälle auftreten
- ... besser angepasster Code erzeugbar
- Beispiele: Konstante Operanden
 - $x := y * 4 \rightarrow x := y \ll 2$
 - $x := 17 * 4 \rightarrow x := 68$
- Weitere Beispiele
 - Schlüsselloch-Optimierung (*peephole optimization*)
 - `ST r1, (0x400); LD r2, (0x400); ADD r3, r2`
→
 - `ST r1, (0x400); ADD r3, r1`
 - Ersetze Tail-Recursion durch Sprung
 - Verwendet (modifizierten) alten Stack Frame wieder

1. Versuch (aus Block 4: Datenfluss)

$\text{CONSTANTS}(b)$ sind alle bisher gesammelten Aussagen zu Beginn des Blocks b

A. Koch

- Keine Aussage über v machbar: $(v, x) \notin \text{CONSTANTS}(b)$
- v ist konstant mit Wert c : $(v, c) \in \text{CONSTANTS}(b)$
- v hat unbekanntem (potentiell variablen) Wert: $(v, \perp) \in \text{CONSTANTS}(b)$

Definition Meets-Operator

$$(v, c_1) \wedge (v, c_2) = \begin{cases} (v, c_1) & : \text{wenn } c_1 = c_2 \\ (v, \perp) & : \text{sonst} \end{cases}$$

```
i0 := 12;
while ( ... ) {
    i1 := phi (i0, i3);
    x1 := i1 * 17;
    j1 := i1;
    i2 := ...;
    i3 := j1;
}
```

1. Version rechnet bei Join-Knoten: $i_0 \wedge i_3 = 12 \wedge \perp = \perp$

Programmausführung liefert aber anderes Ergebnis!

1. Version ist **pessimistisch**: Verknüpfung mit unbekanntem Wert liefert immer \perp

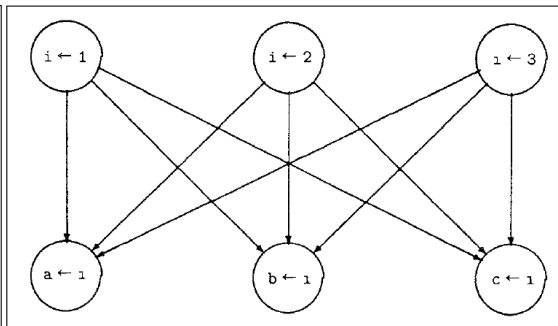
- In SSA-Form: *Sparse*
 - Jeder Wert hat genau **eine** Quelle
- Ohne Berücksichtigung von Kontrollfluss: *Simple*

➔ *Sparse Simple Constant Propagation (SSC, SSCP)*

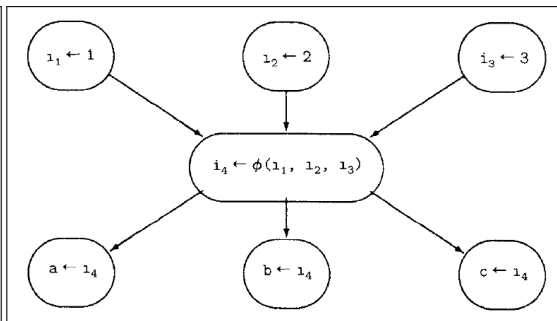
Programmbeispiel Def-Use-Graph

A. Koch

```
select j
  when x {i ← 1}
  when y {i ← 2}
  when z {i ← 3}
end
select k
  when x {a ← i}
  when y {b ← i}
  when z {c ← 1}
end
```



```
select j
  when x {i ← 1}
  when y {i ← 2}
  when z {i ← 3}
end
select k
  when x {a ← i}
  when y {b ← i}
  when z {c ← i}
end
```

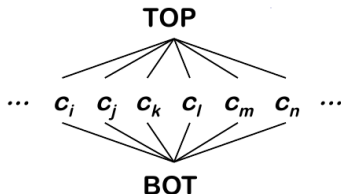


A. Koch

- Weniger Kanten (dünner oder spärlich besetzt, *sparse*)
- Vorsicht: Verschiedene Arten von SSA-Graphen
 - Unterscheiden sich in Richtung der Kanten
 - Hier: Def→Use

- **Optimistischer** Ansatz
- Variablen werden als potentiell konstant angenommen, solange nicht das Gegenteil bewiesen ist
- Darstellung der Fakten als Verband
 - \top (TOP, UNDEF): Noch nichts bekannt, Variable ist potentiell konstant
 - c : Variable hat den konstanten Wert c
 - \perp (BOT, NAC, not-a-constant): Variable ändert sich

A. Koch



Transferregeln für Fakten bei Anweisungen $\mathbf{x} := \mathbf{y} \text{ op } \mathbf{z}$

A. Koch

Grundsätzlich: $\text{Value}(\mathbf{x}) = \dots$

1. $c_1 \text{ op } c_2 = \text{Auswertung von } \text{op}$, falls $\text{Value}(\mathbf{y}) = c_1$,
 $\text{Value}(\mathbf{z}) = c_2$
2. \perp , falls $\text{Value}(\mathbf{y}) = \perp$ oder $\text{Value}(\mathbf{z}) = \perp$
3. \top , sonst

Erweiterung: Rechnen mit Null-Elementen bei $y \in \{\perp, \top\}$

$$y \cdot 0 = 0 \cdot y = 0$$

$$y \text{ AND false} = \text{false AND } z = \text{false}$$

$$y \text{ OR true} = \text{true OR } z = \text{true}$$

Aufeinandertreffen von Fakten an Join-Knoten:

$$\text{Value}(\Phi(v_1, v_2, \dots, v_n)) = \text{Value}(v_1) \wedge \text{Value}(v_2) \wedge \dots \wedge \text{Value}(v_n)$$

Optimistischer Meets-Operator

\wedge	\top	c_1	\perp
\top	\top	c_1	\perp
c_2	c_2	$(c_1 = c_2)?c_1 : \perp$	\perp
\perp	\perp	\perp	\perp

Sparse Simple Constant Propagation 4

\forall expression, e $\left\{ \begin{array}{l} \text{TOP} \text{ falls Wert unbekannt ist} \\ c_i \text{ falls Wert bekannt ist} \\ \text{BOT} \text{ falls Wert variiert} \end{array} \right.$
Value(e) \leftarrow —
WorkList $\leftarrow \emptyset$

\forall SSA edge s = $\langle u, v \rangle$
if Value(u) \neq TOP then
add s to WorkList

while (WorkList $\neq \emptyset$)

remove s = $\langle u, v \rangle$ from WorkList
let o be the operation that uses v

o ist "a \leftarrow b op v" oder "a \leftarrow v op b"
oder "a \leftarrow v op v"

if Value(o) \neq BOT then

t \leftarrow result of evaluating o

if t \neq Value(o) then

Value(o) \leftarrow t

\forall SSA edge $\langle o, x \rangle$

add $\langle o, x \rangle$ to WorkList

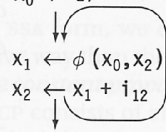
SSA-Graph Kanten: Def \rightarrow Use

Ausführungszeit

- Maximal zwei Value-Änderungen je Variable
 - $\top \rightarrow c$
 - $c \rightarrow \perp$
 - Variablen in Worklist aufgenommen nur bei Änderungen (also max. zweimal)
 - Operation wird evaluiert, wenn einer der beiden Operanden Worklist entnommen wird
- ➔ Max. Evaluationen: 4x Anzahl der Operatoren

Pessimistischer und Optimistischer Ansatz

$x_0 \leftarrow 17$
 \downarrow
 $x_1 \leftarrow \phi(x_0, x_2)$
 $x_2 \leftarrow x_1 + i_{12}$
 \downarrow



Time Step	Lattice Values					
	Pessimistic			Optimistic		
	x_0	x_1	x_2	x_0	x_1	x_2
0	17	\perp	\perp	17	\top	\top
1	17	\perp	\perp	17	17	$17 + i_{12}$
2	17	\perp	\perp	$17 \wedge (17 + i_{12}) \dots$		

Beispiele: $i_{12} = 0$ und $i_{12} = 2$

Was passiert, wenn wir Fakten in eine Sprungbedingung propagieren?

\top : Wir wissen noch nichts

\perp : Beide Pfade können auftreten

TRUE/FALSE : Nur ein Pfad wird ausgeführt

➡ Nur ein Pfad hat einen **Effekt** auf die Programmausführung

➡ Potentielle Effekte anderer Pfade **ignorieren**

- Werte: Von Definition zu Benutzung
 - **SSAWorkList**
- Kontrollfluß: Entlang von Kanten zu **erreichbaren** Blöcken
 - **CFGWorkList**
- Verbreite Werte nur in **erreichbare** Blöcke

Initialisierung

SSAWorkList := \emptyset

CFGWorkList := $\{b_0\}$

\forall Block b

 setze b als unerreichbar

\forall Operation o in b

 Value(o) := \top

Unterschied zu SCP: Nimm zunächst Ergebnisse **aller** Operationen als \top (UNDEF) an

Sparse Conditional Constant Propagation 2

Ausbreitungsregeln: Daten

$x \leftarrow c, c \text{ const.}$

$\text{Value}(x) := c$

A. Koch

$x \leftarrow \phi(y, z)$

$\text{Value}(x) := \text{Value}(y) \wedge \text{Value}(z)$

wie bei SSCP

$x \leftarrow y \text{ op } z$

if $\text{Value}(y) \neq \perp$ **and** $\text{Value}(z) \neq \perp$ **then**

$\text{Value}(x) := \text{Value}(\text{"}y \text{ op } z\text{"})$

wie bei SSCP

Immer:

Falls $\text{Value}(x)$ geändert

$\forall (x, o) \in \text{SSA-Graph}$

// Kanten: Def \rightarrow Use

falls $\text{block}(o)$ erreichbar

$\text{SSAWorkList} := \text{SSAWorkList} \cup \{(x, o)\}$

Sparse Conditional Constant Propagation 3

Ausbreitungsregeln: Kontrollfluß

branch $cond, l_t, l_f$

if $cond \in \{\perp, TRUE\}$ **and** Block l_t unerreichbar **then**
CFGWorkList := CFGWorkList $\cup \{l_t\}$

if $cond \in \{\perp, FALSE\}$ **and** Block l_f unerreichbar **then**
CFGWorkList := CFGWorkList $\cup \{l_f\}$

jump l

if Block l unerreichbar **then**
CFGWorkList := CFGWorkList $\cup \{l\}$

while ((CFGWorkList \cup SSAWorkList) $\neq \emptyset$)

while (CFGWorkList $\neq \emptyset$)

nimm einen Block b aus CFGWorkList

markiere b als erreichbar

// benutze Rechenregeln

evaluiere \emptyset -Funktionen in b , parallel

evaluiere Operationen in b , in Programmreihenfolge

while (SSAWorkList $\neq \emptyset$)

nimm eine Kante $s = \langle u, v \rangle$ aus SSAWorkList

es sei o die Operation, die die Verwendung v enthält

// benutze Rechenregeln

evaluiere Operation o

- Sprungbedingungen auf \top sollten nicht auftreten
 - Compiler-Fehler?
- Alle Operationen auf \top initialisieren
 - Kontrollfluß läßt Werte $\neq \top$ zu
 - Unerreichbare Pfade tragen \top zu **optimistischen** Phi-Funktionen bei
- Hier Vorschlag: Erst CFG-Kanten, dann SSA-Kanten
 - Könnte aber in beliebiger Reihenfolge geschehen
 - CFG-Kanten zuerst kann aber etwas schneller sein

- Auch hier: Null-Elemente in Rechnungen berücksichtigen
 - $\top * \perp \rightarrow \top$
 - Grund: Falls $\top \rightarrow 0$, ist $0 * \perp \rightarrow 0$
 - Analog: AND, OR, etc.
- Auch variable Werte können zu Vereinfachungen führen
 - $\perp * c \rightarrow \perp$, nicht konstant
 - Kann bei z.B. $c = 2$ zu Vereinfachung führen (Shift)
 - Aber Nebeneffekt: Dann nicht mehr kommutativ!
- Hier nicht gezeigt: Umschreiben von Verzweigungen zu Sprüngen
 - **branch TRUE, L1, L2 \rightarrow jump L1**


```
i ← 17
if (i > 0) then
  j1 ← 10
else
  j2 ← 20
j3 ← Ø(j1, j2)
k ← j3 * 17
```

Beispiel SCCP

Ohne Berücksichtigung von Kontrollfluss (SSCP)

```
17  i ← 17
    if (i > 0) then
10   j1 ← 10
    else
20   j2 ← 20
    ⊥  j3 ← ∅(j1, j2)
    ⊥  k ← j3 * 17
```

All paths
execute

Beispiel SCCP

Mit Berücksichtigung von Kontrollfluss (SCCP)

```
17  i ← 17
    if (i > 0) then
10  j1 ← 10
    else
TOP  j2 ← 20
10  j3 ← Ø(j1, j2)
170 k ← j3 * 17
```

With SCC
marking
blocks

A. Koch

Effekt kann durch DEAD nicht erreicht werden

- DEAD kann nicht $i > 0$ evaluieren
- Damit ist $j_2 \leftarrow 20$ eine nützliche Anweisung

➔ Kombination von Optimierungen kann sinnvoll sein

Grundlagen, Ausdehnung auf Inter-Prozedur-Bereich

M.N. Wegman & F.K. Zadeck

Constant propagation with conditional branches

ACM TOPLAS, 13(2), April 1991, Seiten 181...210

A. Koch

Vertiefung, andere Notation

C. Click & K. D. Cooper

Combining Analyses, Combining Optimizations

ACM TOPLAS, 17(2), März 1995, Seiten 181...196

Beide Papers auf OC Web-Seite.

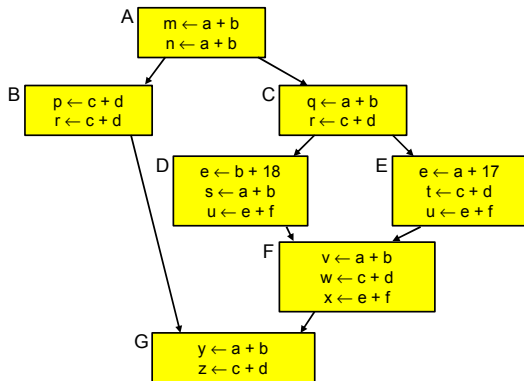
Optimierungsermöglichende Transformationen

- Führen selbst noch keine Optimierung aus
- Können Code sogar größer/komplizierter machen
- Bieten dann aber mehr Angriffspunkte für andere Optimierungen

Vervielfältigen von Blöcken 1

Cloning – Vorher

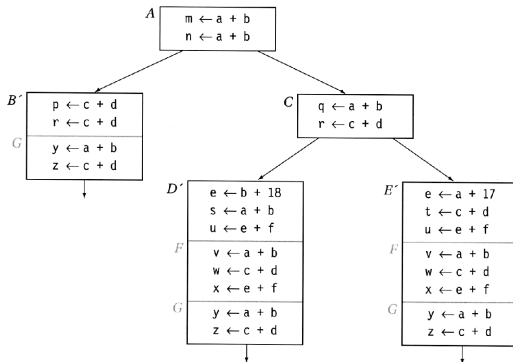
A. Koch



Vervielfältigen von Blöcken 2

Cloning – Nachher

A. Koch



Vervielfältigen von Blöcken 3

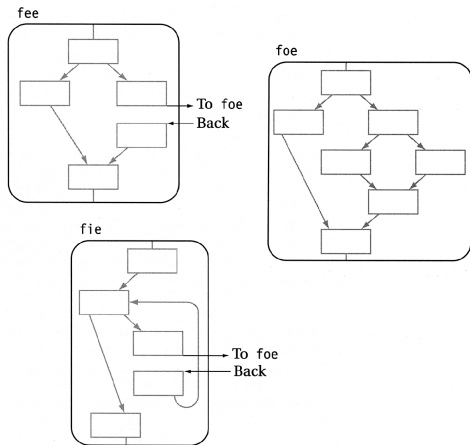
Diskussion

- Eliminieren von Merge-Points
- ... durch Kopieren und Zusammenfügen von Blöcken
- Vorteile
 - Längere Blöcke, mehr Kontext für lokale Verfahren
 - Beseitigen von Verzweigungen
 - Mehr Ansatzpunkte für Optimierungen
- Nachteile
 - Mehr Code, potentiell mehr I\$-Misses

Inlining von Unterprogrammen 1

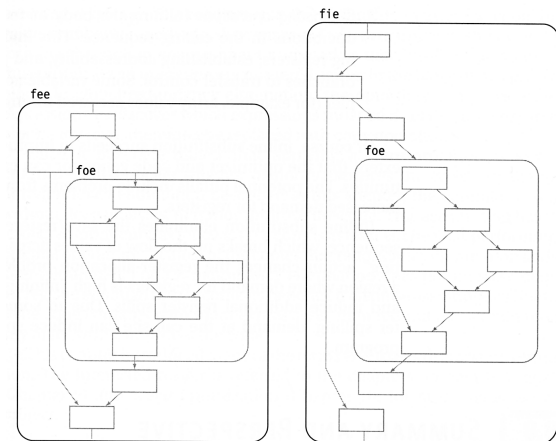
Vorher

A. Koch



Inlining von Unterprogrammen 2

Vorher



A. Koch

Inlining von Unterprogrammen 3

Diskussion

- Vorteile
 - Code des Unterprogrammes nun im Kontext des Aufrufers → Mehr Ansatzpunkte für Optimierung
 - Aufrufsequenz (Parameterübergabe, Stack Frame, Sprung, etc.) entfällt
- Nachteile
 - Mehr Code, potentiell mehr I\$-Misses und höhere Registernachfrage

Abrollen von Schleifen 1

Loop Unrolling

Vorher

```
do i = 1 to n by 1
  a(i) = a(i) + b(i)
end
```

Nachher

Annahme: n bekannt

```
do i = 1 to 100 by 4
  a(i)   = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
  a(i+3) = a(i+3) + b(i+3)
end
```

Abrollen von Schleifen 2

Loop Unrolling

Nachher

Annahme: n unbekannt

```
i = 1
do while (i+3 ≤ n)
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  a(i+2) = a(i+2) + b(i+2)
  a(i+3) = a(i+3) + b(i+3)
  i = i + 4
end
do while (i ≤ n)
  a(i) = a(i) + b(i)
  i = i + 1
end
```

Nachher

Annahme: n unbekannt

```
i = 1
if (mod(n,2) > 0) then
  a(i) = a(i) + b(i)
  i = i + 1
if (mod(n,4) > 1) then
  a(i) = a(i) + b(i)
  a(i+1) = a(i+1) + b(i+1)
  i = i + 2
do j = i to n by 4
  a(j) = a(j) + b(j)
  a(j+1) = a(j+1) + b(j+1)
  a(j+2) = a(j+2) + b(j+2)
  a(j+3) = a(j+3) + b(j+3)
end
```

- Vorteile
 - Reduzierte Zahl ausgeführter Anweisungen
 - Erhöht Anzahl von Anweisungen im Schleifenrumpf
 - Mehr Möglichkeiten, unabhängige Anweisungen parallel auszuführen
 - Mehr Anweisungen, um Branch Delay Slots zu füllen
 - Mehr aufeinanderfolgende Speicherzugriffe zusammen in Schleife
 - Potential für Vektorisierung (SIMD-Ausführung)
- Nachteile
 - Mehr Code, potentiell mehr I\$-Misses

```
do i = 1 to n
  if (x > y)
    then a(i) = b(i) * x
    else a(i) = b(i) * y
```

Original Loop

```
if (x > y) then
  do i = 1 to n
    a(i) = b(i) * x
else
  do i = 1 to n
    a(i) = b(i) * y
```

Unswitched Version

Weniger Kontrollfluss innerhalb der Schleife

- Weniger ausgeführte Instruktionen
 - Insbesondere potentiell langsame Verzweigungen
- Mehr einfacher zu optimierender “straight-line-code”

Übergreifende Diskussion

Code Hoisting Berechnet VERYBUSY-**Ausdrücke** einmal

- Verlangsamt Programm nicht

A. Koch

Sinking Verschiebt wiederkehrende **Anweisungsfolgen**
im CFG nach vorne

- Code wird nur einmal erzeugt,
- ... aber von mehreren Ästen benutzt

Cross Jumping Prüft Anweisungen **vor** Sprung an ein Label

- Identische Folgen werden hinter Label geschoben

Procedure Abstraction Klammere wiederkehrende
Anweisungsfolgen in neue Prozedur aus

- Falls ausgeklammerter Code größer als Aufrufsequenz
- ... Platz gespart
- Verlangsamt Programmausführung

Procedure Placement Häufig aufgerufene Prozeduren in der Nähe des Aufrufers plazieren

- Idealerweise gleicher Speicherseite

Block Placement Häufig genommene Verzweigungen als Fall-Through realisieren

- Vermeidet langsamen Sprung
- Nutzt I\$-Prefetching besser aus
- Beispiel für Profile-basierte Optimierung

Fluff Removal Bewege selten benutzten Code an den Rand des Programmes

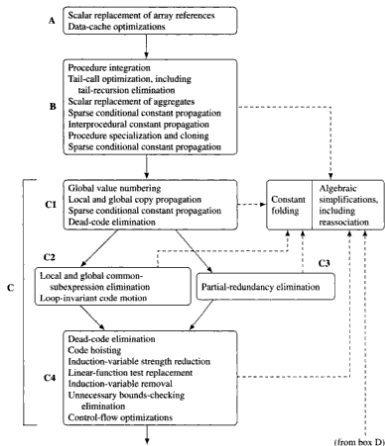
- Steigert die I\$-Cache Effizienz (gecachte Anweisungen werden i.d.R. benutzt)

Schwieriges Problem!

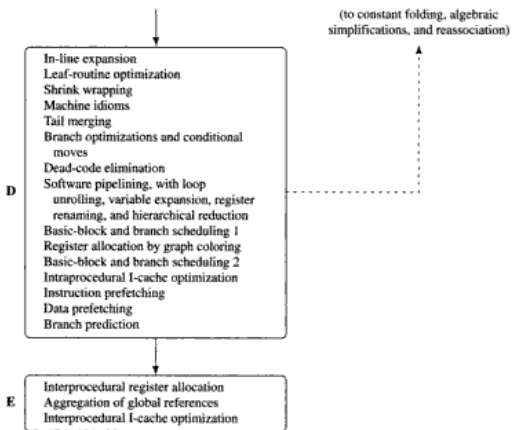
A. Koch

- Alternative Ansätze
 - GCSE oder LCM oder PRE?
- Interaktionen zwischen Optimierungen
 - Verstärkend
 - Constant Propagation / LCM / PRE verbessern Operator Strength Reduction
 - Verschlechternd
 - Redundanzeliminierung verlängert Lebenszeiten
 - ... damit Registervergabe schwieriger
 - Überlappend
 - Constant Folding in Wertnumerierung
 - SCCP

Vorschlag einer Optimierungsreihenfolge 1



Vorschlag einer Optimierungsreihenfolge 2



Zusammenfassung

- Erster Einblick in skalare Optimierung
- Modifikation des CFG
- Neue und alte Konzepte
 - Dominanz, Postdominanz, Dominanzgrenzen
- Dead Code Elimination: DEAD
- Bereinigen des CFG: CLEAN
- Spezialisierung: SSCP, SCCP
- Ermöglichende Transformationen
- Übergreifende Diskussion